A FORTRAN PROGRAM STORAGE AND RETRIEVAL PACKAGE FOR AUTOMATIC MANIPULATION SYSTEMS

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

B. YUGANDHAR

to the

COMPUTER SCIENCE PROGRAM
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY 1977

CSP-1977-M-YUG-FOR

CERTIFICATE

THIS is to certify that the thesis entitled, 'A FORTRAN PROGRAM STORAGE AND RETRIEVAL PACKAGE FOR AUTOMATIC MANIPULATION SYSTEMS', is a record of the work carried out under my supervision and that it has not been submitted elsewhere for a degree.

H.V. Sahasrabuddhe

July 5, 1977

(Dr.) H.V. Sahasrabuddhe
Assistant Professor
of
Electrical Engineering and
Computer Science
Indian Institute of Technology, Kanpur

ACKNOWLEDGEMENTS

It is with great pleasure and deep sense of gratitude that I acknowledge my gratefulness to my thesis supervisor, Dr. H.V. Sahasrabuddhe, for his inspiring guidance, constant encouragement, and useful suggestions. He was always willing to help me at any time during the course of this work and he devoted many of his precious hours for long sessions of discussions.

I thank Mr. B.H. Jajoo for his help in this work. My thanks are also due to my friends and colleagues but for whom my stay here would not have been so pleasant and memorable.

Thanks are also due to Mr. H.K. Nathani for his elegant and neat typing.

\- B. Yugandhar

Kanpur
July 5, 1977

CONTENTS

ABSTRACT

A FORTRAN program package, which can read a FORTRAN program segment and store it in the form of a flowgraph and which can convert the flowgraph back into FORTRAN program, was designed and implemented. The package essentially processes one statement at a time until end of the program segment is reached. This package can become a part of a large system, using which automatic program manipulations can be achieved.

-

CHAPTER 1

INTRODUCTION AND MOTIVATION

1-1. INTRODUCTION

Automatic program manipulation techniques will greatly simplify the problems of a user, who wants to write programs. He can save a lot of time, thus enabling him to focus his attention on more creative aspects of programming. Examples of manipulations on a program are optimisation, block identification, data flow analysis, structure refining etc., of the given program [Baker, 1977, Lowry and Medlock, 1969, Standish, et. al., 1976, and Schneck and Angel, 1973].

Optimisation of a program is in terms of execution speed improvements. Given any source program, we should be able to manipulate it to come out with a highly optimised equivalent program. Thus the task of writing programs becomes easy for a user as he is relieved of writing efficient code. We can say that the advantage of above manipulation is that it transfers the lengthy, error prone, and essentially clerical task of source code optimization from the programmer to the computer.

Block identification of a program is also useful as one can see the control flow of the program. Various loops present in the program etc., can be identified easily. Data flow analysis of a program gives out details of different

variables like where they are defined and where they are used etc. For example, given a program point, by means of this manipulation we can get information about what data definitions are 'live' at that point, that is, what data definitions given before this point are used after this point. So a user can be informed that a variable has been defined but never used, perhaps an indication of a typographical error.

Having seen that automatic program manipulation techniques are indeed helpful to a user, we now see how they can be achieved. The basic needs to achieve them are as follows.

1. The program should be stored in such a format that it is suitable for efficient manipulation.
2. Control flow information of the program should be easily available.
3. Data flow in the form of symbol table(s) should be available.

Storing the given program in the form of flowgraph, and maintaining different symbol table(s) will satisfy the above requirements. In this project a FORTRAN program package, which can read a FORTRAN program segment and store it in the form of flowgraph, and which can convert the flowgraph back into FORTRAN program was designed and implemented.

FORTRAN was chosen as source language because in this computer centre, FORTRAN is widely used and so this package will be useful. To make the system written machine independent, it was written entirely in ANSI standard FORTRAN.

1-2. OVERVIEW OF CHAPTERS

Chapter 2 describes the strategy adopted in the design and implementation of the package. Chapter 3 describes the way in which source program is stored and overall organization of the system. The various routines of the system are explained in Chapters 4, 5 and 6. Chapter 7 gives the future work that can be done and conclusions. This is followed by bibliography and appendix.

CHAPTER 2

STRATEGY ADOPTED

As the hardware techniques are improving, the hardware cost of computers is decreasing very rapidly. But unfortunately the same is not the case with software. The high cost of programming is largely due to the complexity of programs. As a result of this complexity the program development process is marked by large number of mistakes and great deal of waste and rework. Large programming projects in the past have reported coding rates of two to three statements per man-day. Since it would hardly take ten minutes to write three statements, it is very much clear that a lot of time was being wasted in debugging and recoding parts of the system. New techniques like top-down ddsign greatly reduce this waste. We discuss about this later in the chapter.

There are two other important factors which require special emphasis. They are software maintenance and modification. These account for substantial portion of total software expenditure. A program which is easy to read and understand will greatly decrease the cost of program development and maintenance. We tried to develop programs which are less complex and are easy to read and understand. Some of the techniques used to achieve these are as follows.

2-1. TOP-DOWN DESIGN

Top-down design technique was adopted in the design for the following reasons. In top-down design, program development begins at the top most functional level and proceeds decrementally to the lowest functional level. The basic function is broken down into more detailed subfunctions. The process is continued until all sub-functions are defined to a consistent level of detail. Top-down design provides for orderly logic development and reduces the complexity of the programs that result from the design. When we are finished with top-down design process, we will know about all of our interfaces and logic decisions.

2-2. MODULE DIVISION

Parnas' principle of information hiding [Parnas, 1972] was used in deciding the modules of the system. Each module was designed to hide a design decision which is likely to change. In this way only one module will be affected if we want to change a design decision. So using this idéa a data structure, its internal linkings, accessing procedures and modifying procedures become a part of a module. Also character codes and similar data were hidden into a module for greater flexibility. Storage requirements of a data structure were also hidden in modules, so that if need arises they can be manipulated easily.

2-3. AN IMPLEMENTATION TECHNIQUE

A technique used in the implementation stage of the system is as follows. All the programs were first written in a pseudo computer language before coding them in FORTRAN. Most of the control constructs and other features of Pascal [Wirth, 1973] were included in this pseudo computer language. Some of the features of the language are as follows.

Basic symbols:

letters	a ... z
digits	0123456789
arithmetic operators	+ - * / ↑
logical operators	∨ ∧ ¬ ; ∈, ∉
relational operators	= ≠ < ≤ > ≥
parentheses	( )
statement brackets	begin end
assignment operator	←
quote mark	'
seperator	;
jump operator	go to label

Compound statements:	begin S1; S2; ..., Sn end
Conditional statements:	if B then S1 else S2 fi
	if B then S1 fi
Repetitive statements:	while B do S od
	repeat S until B
	for variable = initial value to final value do S od
Selective statements:	case class of L1:S1; L2:S2; ..., Ln:Sn end
Comment statements:	comment:text;

The features were so chosen that the program written in this language is easy to read and understand. Go to statements are used only when there is no better way to describe the flow of control. Much importance was not given to the syntax of the language since it is sufficient if we

understand the logic of the program written.

So all the programs were first written in this pseudo computer language and they were thoroughly read to find any logical errors. In fact errors were detected at this stage only and they were corrected. Thus we could correct most of the errors even before coding them in FORTRAN, thus saving a lot of computer time. Using this technique we could debug a system consisting of about 85 routines in about ten days.

Using the above said techniques, the system was designed and implemented.

CHAPTER 3

DETAILS OF DESIGN

In this chapter we describe how the given FORTRAN program is stored in the form of flowgraph and the overall organisation of the package. In Section 1, the way in which FORTRAN program is represented in the form of flowgraph and the necessary data structures required to store the necessary information are discussed. In Section 2 overall organisation and main components of the package designed are given.

3-1. REPRESENTATION OF FORTRAN PROGRAM AS FLOWGRAPH

The given FORTRAN program is to be represented in such a form that it can be manipulated easily. To accomplish this, the FORTRAN program is broken into 'basic blocks' [Schnek, 1973] whose relationship may be represented by a directed graph that illustrates the flow of control through the program.

A basic block is a set of statements with a single entry point and a single exit point. This means that one can only branch to the first statement of the set of statements of basic block and only the last statement of the set contains a branch to one or more basic blocks. It follows from the above definition of the basic block, that before a given statement of a basic block is executed, all statements preceding it must have been executed. In other words all statements of a basic block are executed sequentially from entry to exit.

Basic blocks are made of a number of successive executable statements, limited by following rules.

A basic block begins,

(1) if a statement number occurs i.e., a statement with statement number is seen. In this, format statement numbers are to be excluded.

(2) after logical IF, arithmetic IF, all types of GO TO statements, DO statement, and STOP/RETURN statement.

A basic block ends,

(1) immediately before a statement having statement number. In this also we have to exclude format statement numbers.

(2) with statements like logical IF, arithmetic IF, all types of GO TO statements, STOP/RETURN statement, and end of range of a DO statement.

However, logical IF statements produce two basic blocks. The statement following the logical expression will form a seperate basic block. DO statements are tackled as follows. DO statement is converted into corresponding logical IF statement. An example makes this clear.

```
    DO 12 I = 1,10                      I = 1
    X = Y                          -1 CONTINUE
                                      X = Y
      :                                 :
                      ====>
    GO TO 12                          GO TO 12
      :                                 :
12  P = Q                          12 P = Q
                                      I = I+1
                                      IF(I.LE.10)GO TO -1
```

Successive negative integers from -1 onwards are assigned to successive DO statements in their order of appearance. For easy recognition of DO statements, negative statement numbers are given to them.

All declarative statements are stored in a separate block called declarative block. This causes all declarative statements to be together in keeping with the ANSI standard.

The control flow information and the set of statements of a basic block are stored as follows.

The information pertaining to a basic block is stored in two blocks called flow block and code block. The flow block contains information which points to a list of all blocks that could be executed immediately after this block. Such blocks are called '<u>successors</u>' of a given block. Also the flow block points to the corresponding code block which contains the set of statements pertaining to this block. It will also have the information regarding the type of segment (main or subroutine etc.) to which the block belongs and the statement number which starts this block. The exact data structure of flow block is as follows:

```
Type:
Stmt. No.
Pointer to code block
Pointer to successor block
Pointer to successor block
:
:
Pointer to successor block
```

As mentioned earlier, the code block contains the set of statements of the corresponding basic block. All the statements are stored as a string, separated by end markers.

In addition, different tables are maintained to store the necessary information. For example, a simple variable when first encountered will be entered into a simple variable table. Similar is the case with others. The different tables that are maintained are -

1. Comment and Format table
2. Constant table
3. Dimensioned variable table
4. Simple variable table
5. Subprogram/function name table
6. Statement number table.

In the code block, all the statements are stored in terms of entry numbers of the corresponding tables. This makes the code block easy to manipulate and offers a saving of storage also. The exact data structures of all the tables are as explained below.

<u>Comment and Format Table:</u> In this table, all comment statements and format statements are stored. Also the class, which indicates comment or format, and the length of the statement in terms of number of characters are stored. The data structure is as shown below.

0	1	2	74	75
Class	Length	Statement	Continuation mark	

If the length of the statement exceeds 72 characters, then a 1 is put in continuation mark location and the rest of the statement stored in the next entry.

Constant Table: In this table, all constants (real, integer, and hollerith constants) are stored. This table also contains the class of constant and the length of the constant. The data structure is as shown below.

0 Class	1 Length	2 Constant	19 Conti-nuation mark 20

If the length of constant exceeds 17 characters, then a 1 is stored in 'continuation mark' and rest of constant stored in the next entry.

Dimensioned Variable Table: In this table all dimensioned variables are stored along with necessary details. The data structure of the table is as shown below.

0 Length	1 Name	7 Exptyp. bit	8 type	9 No. of arg.	10 arg. 13

exptyp bit = 1, if mode of var. has been declared

explicitly.

= 0, otherwise.

If made has been delcared explicitly, then

typcl = 0 for integer

= 1 for real

= 2 for logical

= 3 for complex

= 4 for double precision

No. of arg = 1, if 1-D array

= 2, if 2-D array

= 3, if 3-D array

Variable arg. (1:3) contains the maximum arguments declared, in terms of the entry points in the constant table.

Simple Variable Table: In this table all simple variables, with necessary details are stored. The data structure is as shown below.

0 1 7 8 9

Length	Name	exptyp bit	typcl.

exptyp bit = 1, if mode of var. has been declared explicitly

= 0, otherwise.

If mode has been declared explicitly, then

typcl = 0 for integer

= 1 for real

= 2 for logical

= 3 for complex

= 4 for double precision.

<u>Subprogram/Function Name Table:</u> In this table; names of all subprograms, function names are stored with necessary details. Data structure is shown below.

0	1	7	8	17	18	19 20
Length	Name	No. of arg.	arg.	cont. mrk.	def. bit	def. ent.

Variable No. of arg. gives how many arguments a subroutine or function has and arg contains all the arguments in terms of entry numbers in tables. If the number of arguments exceed nine, then cont. mrk. is made 1 and the rest of arguments stored in the next entry.

def. bit = 0, if subroutine

= 1, if statement function

= 2, if function subprogram

= 3, if declared in external.

The variable defent is defined only for statement functions and points to location where statement is actually stored.

Statement Number Table: All statement numbers are entered into this table. The data structure is as follows:

Statement number (0–1)	fmtflg (1–2)	link (2–3)

fmtflg = 1, if format statement number

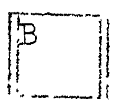
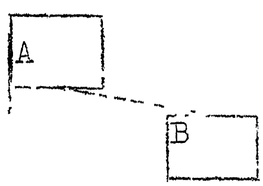
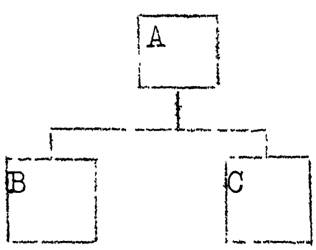
= 0, otherwise.

For format statement numbers, the link points to the entry of format statement in comment and format table. Otherwise, the link points to the flow block corresponding to this statement number.

3-2. OVERALL ORGANISATION AND MAIN COMPONENTS

The overall organisation of the package is given in terms of modified structured charts [Stevens, et. al., 1974]. These charts were chosen as they describe program functions from the topmost level to great detail and the charts also serve as final programming documentation. These charts show how each function is divided into sub-functions.

Only main components of the package are described here. Some of the definitions of symbols used in the charts are as follows.

Symbol	Definition
	module
	predefined module
	Module A invokes module B
	Module A invokes modules B and C. Where possible modules are placed left to right in likely order of invokation.

3-21. Organisation of the Program which Converts a Given FORTRAN Program into Flow-graph

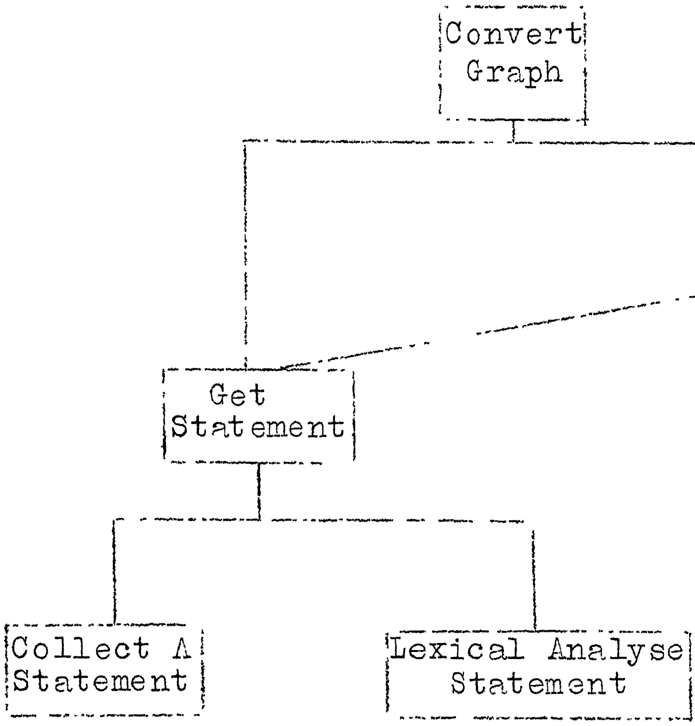


Figure 3-1.

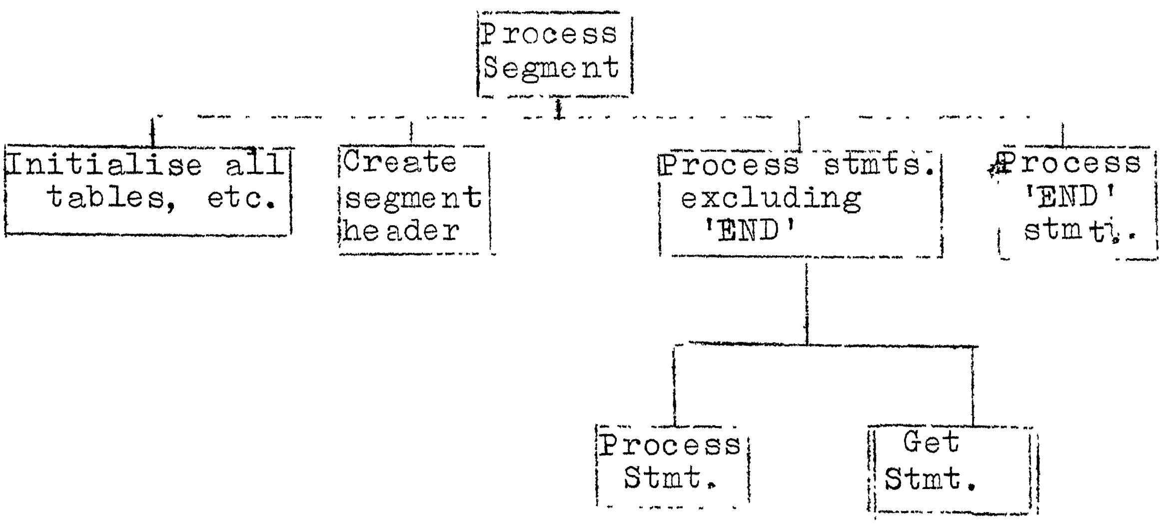


Figure 3-2.

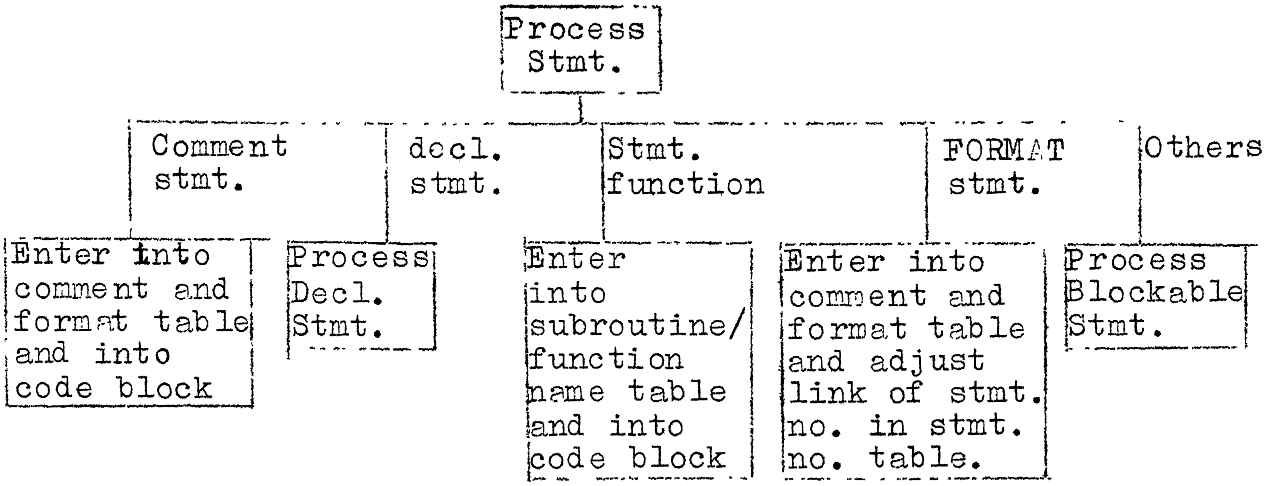


Figure 3-3.

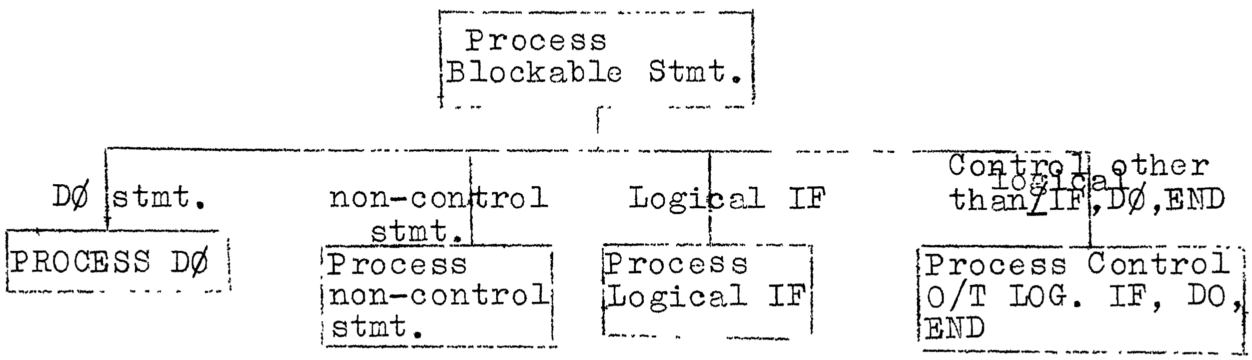


Figure 3-4.

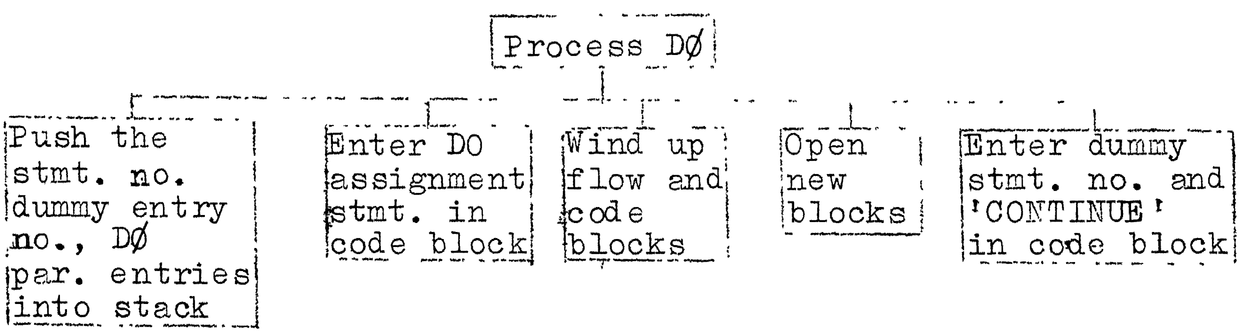


Figure 3-5.

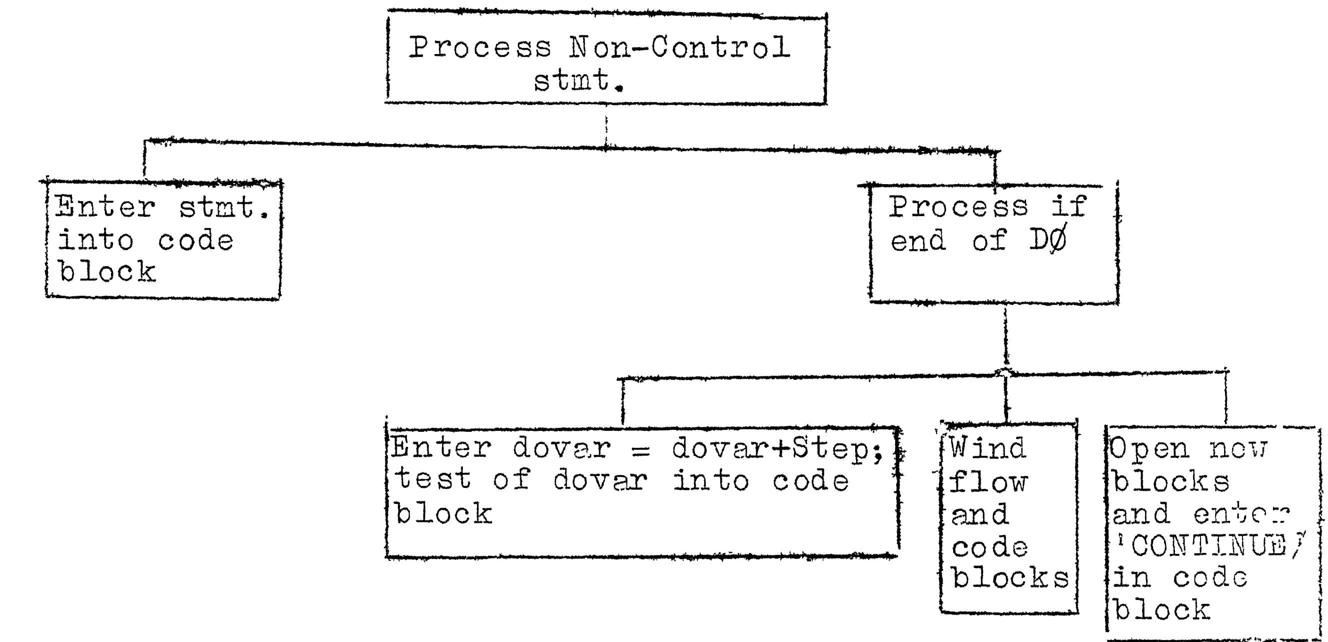


Figure 3-6.

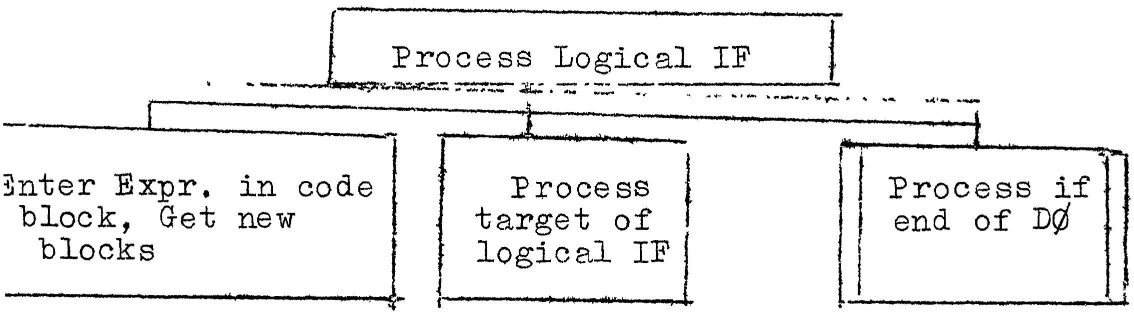


Figure 3-7.

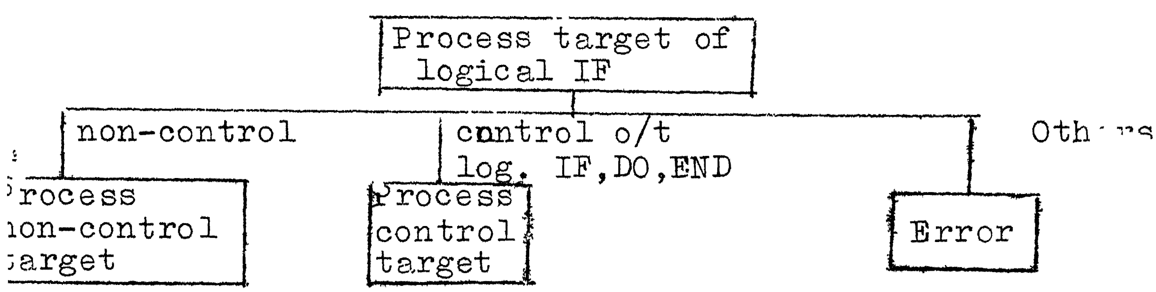


Figure 3-8.

3-22. Organisation of prog. which converts flow-graph into FORTRAN program.

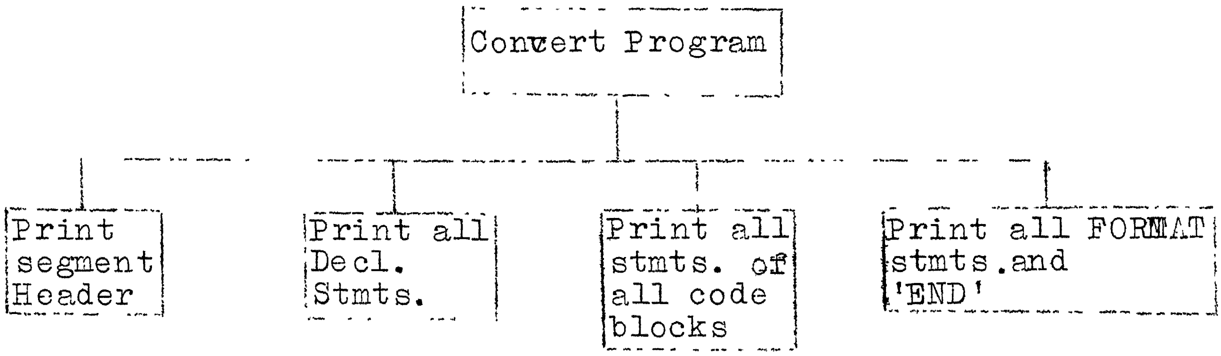


Figure 3-9.

CHAPTER 4

DETAILS OF SUPPORTING ROUTINES

In this chapter the details of all supporting routines, which are used by input part which stores given FORTRAN program in the form of flow graph, and output part which prints out FORTRAN program from the information collected from flowgraph, are given. Some of the routines are used by both parts. Each routine is explained in detail so that one can easily understand the working of it. Parameters, both input as well as output, of each routine, and their functions are also explained.

4-1. OVERVIEW OF DIFFERENT SECTIONS

In Section 2, the routine collect a statement is explained. In Section 3, the lexical analyser and in Section 4 all table handling routines are explained. In Section 5, routines dealing with flow, code, and declarative blocks are explained.

4-2. COLLECT A STATEMENT

A statement from the input program is collected by the routine, called 'clstmt'. The routine alongwith its parameters is

clstmt (stmt(1:700),temp(1:72),endfle,
lnstmt, error)

In this procedure the input parameter is temp(1:72), and output parameters are stmt(1:700), temp(1:72), endfle, lstmt, and error. The significance of each parameters is as follows:

When this procedure clstmt is called, a card from the input is already present in variable temp(1:72). First the procedure transfers this card into stmt(1:700). If this is a comment card then it returns control, after reading next card into temp(1:72). Otherwise it reads next card into temp(1:72) and checks for a continuation mark in column 6, if it is not a comment card. If it is continuation card, then this card is appended to previous card in stmt(1:700) and the above procedure repeated. Only 9 continuation cards are collected. At the end of a statement, a marker is put. The variable lnstmt gives the length of statement in terms of the number of characters. The logical variable endfle will be true if slashes in columns 1 and 2 are encountered in card read into temp(1:72). This indicates end of input deck. The integer variable error will be non-zero if there is some error in the card. For example, if column 6 of first card of a statement is not blank or zero, then error is set to some non-zero value, and this column is taken as blank and proceeded.

So when this procedure returns control, a card after the collected statement will be present in temp(1:72).

4-3. LEXICAL ANALYSER

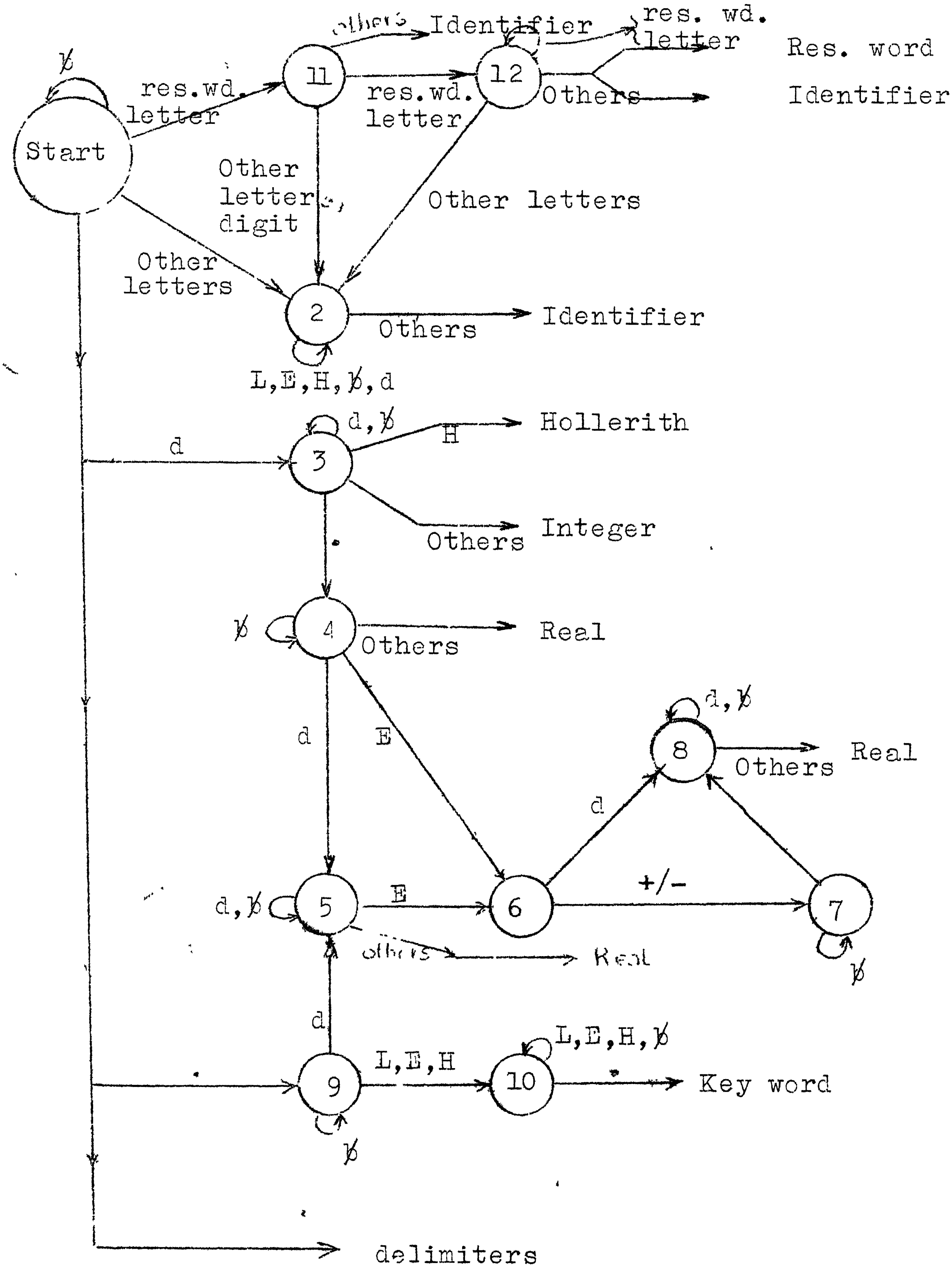
This module takes a statement and lexical analyses it and outputs the tokens. The tokens are stored in koutlx(1:1000). Alongwith each token, its class and length are also stored. The classes of different tokens are given in appendix. Finite state machine technique [Johnson, et. al., 1968] is adopted

to lexical analyse the given statement. We divided the input alphabet into various classes. The input alphabet is divided into 9 groups each one consisting of blank, letters excluding E and H, E, H, digits, + / -, *, . , and other characters respectively. The major class of each group and individual classes of each character are given in the appendix. The state diagram of finite state machine used is shown in Figure 4-1.

To make the state diagram of finite state machine simple, some paths are omitted. When there is no path from a state for a particular charact§r, then it means that there is some error in the input statement. It is seen from the diagram, that while constructing an identifier itself it is seen whether it can form a reserved word. In the input statement, identifiers starting with reserved words are not allowed.

The lexical analyser portion is described as follows. First the various routines, which are used in lexical analysing, are discussed and then the routine which lexical analyses is discussed.

The finite state machine used to lexical analyse a statement is given on the next page. (Figure 4-1).



Start
ƀ
res.wd.
letter
11
others
Identifier
res. wd.
letter
12
Res. word
Others
Identifier
Other
letters,
digit
Other letters
Other
letters
2
Others
Identifier
L,E,H,ƀ,d
d
3
d,ƀ
H
Hollerith
Others
Integer
.
4
ƀ
Others
Real
d
E
8
d,ƀ
Others
Real
d
5
d,ƀ
E
6
+/-
7
ƀ
others
Real
d
.
9
ƀ
L,E,H
10
L,E,H,ƀ
Key word
delimiters

Figure 4-1.

4-31. Procedure nxtchr (stmt(1:700),i); This routine gives out a character from input statement. The character given out is stmt (i+1). The input parameters for this routine are stmt(1:700) and i. The output parameters are i, char, kodch, klch, klass2, kprev, and kprev2. The variables char, kodch, klch, klass2, kprev and kprev2 are common with routines 'dolxi' and 'lexcal'.

The variable i gives position of last character processed. Variable char contains the next character obtained. Variables kodch, klch, and klass2 contain the internal code of the character obtained, the major class of character, and individual class of character respectively. Variables kprev, and kprev2 contain information similar to that contained by klch, and klass2 but this information pertains to previous non-blank character.

In the procedure first i is incremented and then character got. Then it invokes procedure 'chrcod' to get information pertaining to internal code, major class, and individual class of character got. The control is then returned back.

4-32. Procedure chrcod (char, kodh, klch, klass2): The input parameter to this routine is variable char and output paramters arc variables kodch, klch, and klass 2. Given a character, stored in char, it gives out the internal code, major class, and individual class of character. For each character belonging to input alphabet, its major class, and individual class are present in variables istcls(1:64), and indcls(1:64) at a position depending on its internal code.

In the procedure, first internal code of the character is calculated and using this its major class, and individual class are obtained.

4-33. Procedure Fsmtbl (state, klch, nxtact, nstate): The input parameters to this routine are variables state, and klch, and the output parameters are variables nxtact, and nstate.

The procedure has information regarding finite state machine table in variable mchtbl(1:12, 1:9). The first subscript of this variable should be information regarding state, and second subscript should be information regarding major class.

So given present state and major class (given by variables state, and klch respectively), this routine gives the next state and the information regarding type of action to be taken in next phase. In the finite state machine table, both next state and next act are stored together as a number. The last two digits give next state and remaining digits give next act. The variables nstate, and nxtact contain the information regarding next state, and next act at the end of the execution of this routine.

4-34: Procedure Newoul: There is no input parameter to this routine. The output parameter is variable 'kount', present in common area labelled 'output'.

This routine initialises variable kount to one, making it possible to start storing tokens of a new statement. So before storing any token of a new statement this routine should be called.

4-35. Procedure Newtkn: The input parameter to this routine is variable kount. The output parameters are variables klsptr, koutlx(1:1000), and kount. All these variables are of common area labelled 'output'.

Before a new token is stored, this routine should be called. It saves location where class of token is to be stored. Variable klsptr contains this information. Also it initialises class and length to -1. Variable kount is incremented twice, so that it now points to a location, from where the token is to be stored.

4-36. Procedure addchr (khar): The input and output parameters of this routine are variables khar and kount, and koutlx(1:1000) respectively. Output parameters are part of common area labelled 'output'.

This routine adds a character present in khar to the token present in koutlx(1:1000) in the position pointed by kount. Kount is then incremented so that it points to the next available location which is free.

4-37: Procedure defcls (klas): The input and output parameters of this routine are variables klas, and koutlx(1:1000), output parameter being of common area labelled 'output'. The routine stores the class of present token in the location koutlx(klsptr).

4-3.8. Procedure endtkn: The input parameters to this routine are kount, klsptr, and the output parameter is variable koutlx(1:1000). All these variables are of common area labelled 'output'. When storing of a token is over, this routine is called. It stores the length of the token.

4-3.9. Procedure endoul: There is no input parameter to this routine, but the output parameter is variable koutlx(1:1000) of common area labelled 'output'. This routine puts a marker at the end of all tokens of a statement.

4-3.10. Procedure cnsint(n): This routine constructs an integer from the individual digits of the token stored in koutlx(1:1000). The output parameter is variable n which contains the integer constructed. The variable kount at the end of the execution of this routine points to a location from where we can start storing the next token.

4-3.11 Procedure newlxi: This routine initialises the variable kount, belonging to the common area labelled 'output', to one. Thus it makes possible to extract tokens from koutlx(1:1000).

4-3.12. Procedure cheqlx(endlxi): The input parameters to this routine are variables kount, and koutlx(1:1000) of common area labelled 'output'. The output parameter is logical variable endlxi.

This routine checks whether the tokens of statement are over, in which case the logical variable endlxi will be outputted as 'true'. Otherwise endlxi will be outputted as 'false'.

4-3.13. Procedure fchtkn (class, length, token(1:1000)):
The input parameters to this routine are variables kount, and koutlx(1:1000), which are part of common area labelled 'output'. The output parameters are variables class, length, and token (1:1000).

The routine outputs a token starting from location kount. The token is accompanied by its class, and length.

Now the routines dealing with reserved words are described. The manner in which the reserved words are stored is discussed first. All the reserved words are stored in a list structure, so that it is flexible for any addition or modification. The first letters of all reserved words are stored alphabetically, so that binary search can be applied. The remaining letters of reserved word are stored in the same sequence.

The data structure of first character of reserved word is as follows:

upalt	mchar	downalt	msucc.

The given character is compared with mchar (comparison in terms of internal code)

If char $>$ mchar, then go to downalt and repeat step
$=$ mchar, go to msucc, and compare next char
$<$ mchar, then go to upalt and repeat step.

msucc points to a location where the next letter of res. word is stored.

The data structure for other letters (excluding first letter) of reserved word is as follows:

nchar	alt	succ

Given character is compared with nchar

if char = nchar, then go to succ. and compare next char

$\neq$ nchar, then go to alt and repeat step.

'succ' points to location, where the next letter of reserved word is stored. 'Alt' will be zero if there is no other letter which can form a reserved word.

At the end of each reserved word, special marker is put as shown below.

/		

The 'succ' field gives class of reserved word and any other information.

If 'alt' field is non-zero, then depending on the next character, the reserved word is ended here or the comparison is continued with the 'alt' field character. For example, there can be two reserved words starting with END. They are END and ENDFILE. After first three letters are compared, marker is reached and 'alt' field is non-zero. Now the next character is checked and if it is endmarker, then the reserved word ends here and taken as 'END', else the comaprison of next character is continued with char. of location pointed by 'alt'.

The routines which deal with reserved words are described below.

4-3.14. Procedure restbl: The routine does not have any input parameters. It constructs the reserved word table, by reading all the reserved words along with relevant details. It reads in information into variables mchar(1:20), msucc(1:20), idnalt(1:20), and iupalt(1:20), which form the common area labelled 'Rwfrst', and into variables khar(1:200), malt(1:200), and isux(1:200) which form the common area labelled 'Rwothr'.

4-3.15. Procedure strtwd (char, resflg): The input to this routine is variable char, and common area labelled 'Rwfrst'. The output parameters are logical variable resflg, and variable nptr, which is in common area labelled 'Rw'.

The routine checks the first letter of a token for a possible reserved word. The logical variable resflg will be 'true' if input character present in char matches with first letter of any reserved word. If there is a possibility of forming a reserved word with this letter, then variable nptr points to the location where next character of possible reserved word is stored. Binary search technique is used, as the first letters of all reserved words are stored alphabetically.

4-3.16. Function Procedure kode(m): The input to this routine is character present in variable m. It outputs internal code of this character as present in first six bits of the word.

4-3.17. Procedure Oknxch (char, signal, next, class): Input to this routine is common area of reserved word table, labelled 'Rwothr', and the variable char which contains the character to be checked. The output parameters are variables signal, next, class, and nptr. Variable nptr is of common area labelled 'Rw'.

The routine outputs whether the input character is acceptable for forming reserved word.

The variable signal will be zero if input character is unacceptable. It will be one if character is acceptable. It will be two if end of reserved word is encountered. In case of end of reserved word occurance, the variables next and class give the next action which is to be taken and the class of the reserved word respectively. If the character is acceptable, then variable 'nptr' points to location, where next character of reserved word is stored.

4-3.18. Procedure lexcal (stmt(1:700), i, lnstmt, error): The input parameters are the variables stmt(1:700), i and lnstmt, and common area labelled 'Al'. The output parameters are variables koutlx(1:1000), a part of common area labelled 'output', and error.

The routine outputs tokens of stmt(1:700). The first non-blank character of input statement, alongwith its internal code, major class, and individual class is given as input to the routine. All the tokens are stored in the variable koutlx(1:1000). Along with each token, its class and length are also stored in the format (class, length, token). At the

end of all tokens a marker is put.

The routine identifies different tokens of input statement and stores them in koutlx(1:1000). It distinguishes between a logical IF, and arithmatic IF. Also it identifies statements starting with INTEGER FUNCTION ..., REAL FUNCTION ..., etc. A comment statement is stored as single token. Format specifications also are stored as single token.

4-4. TABLE HANDLING ROUTINES

As already described in the previous chapter, we have different tables which are to be maintained. In this section routines dealing with all tables are explained.

4-4.1 Routines dealing with comment and format table: All the routines dealing with common and format table use a common area labelled 'comfmt', which contains variables kmtbl(1:75,1:5) and kmtptr. In this five entries are allowed and if one wants to increase it then the 'common' card in all the routines has to be changed. The variable kmtptr always points to the next available entry in the table.

(a) Procedure inicmt: This routine initialises variable 'kmtptr' of common area to one, thus making it possible to start entering entries into the comment and format table.

(b) Procedure entlne (class, length, token (1:1000), iniptr, entno, errflg): The input parameters to this routine are the variables class, length, token(1:1000) and iniptr. The output parameters are variable entno, and logical variable errflg.

This routine simply enters class, length and token in the entry pointed by kmptr. The variable iniptr gives the location from which point the token is to be taken from token (1:1000). Accordingly actual length of token entered becomes length-iniptr+1. If the length of token being stored exceeds 72 characters, then a continuation mark is put in 75th location of this entry, and the rest of the token stored in the next entry. Of course, the first two locations of entry contain class, and length of the token being stored. If the table is full, then logical variable errflg is returned as 'true'. The variable entno gives entry number in the table where the token is stored. The value given out in entno will have a digit one appended at the end, signifying that this is an entry number of comment and format table, to actual entry number of the table.

(c) Procedure fchlne (class, length, token(1:1000), entno, errflg): The input parameter to this routine is variable entno. The output parameters are variables class, length, token(1:1000), and errflg. Given an entry number in the table, this outputs the token present in this entry alongwith its class and length. It returns errflg as 'true' if a wrong entry number is given. Variable entno gives entry number.

(d) Procedure endcmt: This routine puts a marker at the end of all entries present in the table. The marker is put in the first location of t he entry.

4-4.2. Routines dealing with constant table: All the routines dealong with constant table use a common area labelled 'const', which contains variables knstbl(1:20, 1:50), and knsptr. We allow only fifty constants to be entered. If one wants more, one has to simply change the 'common' card depending on the requirements. Each entry has twenty locations. The variable knsptr always points to the next available entry in the table.

(a) Procedure inicns: This routine simply initialises variable knsptr to one. After the initialisation, one can start entering the entries into the table.

(b) Procedure ontcns (class, length, token(1:1000), entmo, errflg): The input parameters to the routine are the variables class, length and token (1:1000) and the output parameters are the integer variable entno, and logical variable errflg.

In the constant table integer, real, and hollerith constants are entered. This routine enters a constant present in variable token(1:1000), along with its class, and length. If the length of constant exceeds 17 locations, then a continuation mark is put in 20th location of the entry and the rest of the constant entered in the next entry number. In each entry the first and second locations give the class and length of the constant being present in that entry. Real and hollerith constants are stored in character form whereas integer constant is stored as integer number. If the table is full, then

logical variable errflg is returned as 'true'. This check is done in the beginning only. The variable entno contains the actual entry number, where the constant is stored, and a digit two appended at the end, which specifies that the entry number is of constant table.

(c) Procedure fchcns (class, length, token (1:1000), entno, errflg): The input parameter to this routine is variable entno. The output parameters are class, length and token (1:1000). The variable entno specifies the entry from where to fetch the constant and the routine fetches the constant, along with its class, and length. The constant, its class and length are available in variables token (1:1000), class, and length, at the end of execution of routine.

(d) Procedure endcns: This routine simply puts a marker at the end of all entries in the table. The marker is put in the first location of the entry.

4-4.3. Routines dealing with dimensioned variable table: All the routines dealing with dimensioned variable table use a common area labelled 'dmnson', which contains integer variables dmntbl(1:13,1:50), and dmnptr. The table size is fifty, each entry having 13 locations. The variable dmnptr always points to the next available entry in the table.

(a) Procedure indmn: This routine initialises the variable dmnptr to one, after which we can start entering the entries.

(b) Procedure entdmn (lnth, token(1:1000), exptyp, typcl, entno, errflg): The input parameters to the routine are the variables lnth, token (1:1000), exptyp, and typcl. The output parameters are integer variable entno, and logical variable errflg.

In the dimensioned variable table, all the declared dimenioned variables are entered. This routine enters the name of dimensioned variable from locations 2 to 7 of the entry. The 1st location of the entry contains the length of the name of variable to be stored. The name of variable is present in token (1:1000). The routine also stores value present in variable exptyp in location 8 of the entry. If exptyp is one, then typcl is stored in location 9 of the entry. The routine first enters the entry in the entry pointed by dmnptr and then searches table if the entry is already present. If it is not already present in the table, then dmnptr is incremented by one. If the table is full then the logical variable errflg is returned as 'true'. The routine returns entno, which contains actual entry number where dimensioned variable is stored, and a digit three appended at the end, which specifies that the entry number is of dimensioned variable table.

(c) Procedure adjdmn (entno, numarg, arg(1:5), errflg): The input parameters are entno, numarg, and arg(1:5), and the output parameter is errflg. The variable entno gives the entry number, when the last digit three is removed. Additions

are to be made in this entry. The variable numarg gives the number of arguments dimensioned variable has and arg(1:5) contains the entry numbers of constant table where the maximum arguments of dimensioned variable are stored. This routine stores numarg, and arg(1:3) in the locations ten, and 11-13 respectively depending on the number of arguments. If a wrong entry number is given, then logical variable errflg is returned as 'true'.

(d) Procedure fchdmn (entno, lnth, token (1:1000), exptyp, typcl, numarg, arg(1:5), errflg): The input parameter to this routine is variable cntno. The output parameters are variables lnth, token (1:1000), exptyp, typcl, numarg, arg(1:5), and errflg. The variable entno specifies the entry from where to fetch the dimensioned variable and its details. The routine fetches the length of variable, name of variable, information about its explicit declaration, if declared explicitly then its mode, how many arguments it has, and the entry numbers of maximum arguments declared, into variables lnth, token(1:1000), exptyp, typcl, numarg, and arg(1:5) respectively. If the entry number given is a wrong one, then logical variable errflg is returned as 'true'.

(e) Procedure schdmn (lnth, var(1:10), found, entno): The input parameters to this routine are variables lnth, and var(1:10). The output parameters are variables found, and entno. Given the length and name of a variable, this routine

searches the dimensioned variable table for presence of given name. If it is found in the table, then logical variable found is returned as 'true' and the variable entno is returned with value of entry number where found, after appending digit 3 at the end.

(f) Procedure enddmn: This routine puts a marker at the end of all entries of the table. The marker is put in the first location of the entry.

4-4.4 Routines dealing with simple variable table: All the routines dealing with simple variable table use a common area labelled 'smpvar' which contains variables mpltbl(1:9,1:100) and mplptr. The table size is hundred, each entry having nine locations. The variable mplptr always points to the next available entry in the table.

(a) Procedure inismp: This routine initialises variable mplptr to one.

(b) Procedure entsmp (lnth, token(1:1000), exptyp, typcl, entno, errflg): The input parameters to the routine are variables lnth, token(1:1000), exptyp, and typcl. The output paramters are integer variable entno, and logical variable errflg.

The routine first enters the length, name of variable, information regarding its explicit declaration, if explicitly declared then its mode, which are available in variables lnth, token(1:1000), exptyp, and typcl respectively. This is entered in entry pointed by mplptr. Now the table is searched if the

variable is already present, in which case the variable *mplptr* is not incremented by one. If the table is full, then *errflg* is returned as 'true'. A digit four is appended at the end to the actual entry number and stored in *entno*. Last digit 4 tells us that this entry number is of simple variable table.

(c) *Procedure fchsmp (entno, lnth, token (1:1000), exptyp, typcl, errflg)*: The input parameter to this routine is variable *entno*. The output paramters are variables *lnth*, *token(1:1000)* *exptyp*, *typcl*, and *errflg*. The variable *entno* specifies the entry from where to fetch the simple variable and its details. The routine fetches the length of variable, name of variable, information about its explicit declaration, and if declared explicitly then its mode, into variables *lnth*, *token(1:1000)*, *exptyp*, and *typcl* respectively. If the entry number given is a wrong one, then logical variable '*errflg*' is returned as '*true*'.

(d) *Procedure endsmp:* This routine puts a marker in the first location, at the end of all entries of the table.

4-4.5 *Routines dealing with subprogram/function table:* All the routines dealing with subprogram/function table use a common area labelled 'subfnd', which contains integer variables *sfntbl(1:20, 1:40)*, and *sfnptr*. The table size is 40, each entry having 20 locations. The variable *sfnptr* always points to the next available entry in the table.

(a) *Procedure inisfn:* This routine initialises variable *sfnptr* to one.

(b) Procedure entsfn (lnth, token(1:1000), entno, errflg, flag): The input parameters to the routine are variables lnth and token (1:1000). The output parameters are the variables entno, errflg, and flag.

This routine first enters the length and name of the subprogram/function which are available in variables lnth, and token(1:1000). The entry is made at entry number pointed by sfnptr. The table is then searched to check whether this entry is already present, in which case the variable sfnptr is not incremented by one. If the entry is already present then variable flag is returned as 'true'. If the table is full, then errflg is returned as 'true'. The variable entno is returned with the entry number, with a digit five appended at the end. The last digit 5 tells us that this entry number is of subprogram/function table.

(c) Procedure adjsfn (entno, numarg, arg(1:20), defbit, defent, chkbit, errflg): The input parameters to this routine are variables entno, numarg, arg(1:20), defbit, defent, and chkbit. The output parameter is logical variable errflg.

Given an entry number, this routine either enters the arguments of subprogram and other details or it checks the number of arguments etc., if the details are already present. If chkbit is zero then it does the former and if chkbit is 1 then it does the latter. When chkbit is zero the routine enters the number of arguments, arguments, information saying whether this entry is statement function name, and if so where statement

function definition is stored, which are available in the variables numarg, arg(1:20), defbit, and defent. The arguments stored will be in terms of entry numbers in the tables. If there are more than 9 arguments, then a continuation mark is put in 18th location of entry and the rest of the arguments are stored in next entry. The variable defbit should be 1 in case of statement function and then variable defent points to code block where the definition is stored. If the entry number given is a wrong one, then the logical variable errflg is returned as 'true'.

(d) Procedure fehsfn (entno, lnth, token(1:1000), numarg, arg(1:20), defbit, defent, errflg): The input parameter to this routine is variable entno and the output parameters are variables lnth, token(1:1000), numarg, arg(1:20), defbit, defent, and errflg. The variable entno specifies the entry from where to fetch the subprogram/function name and its details. The routine fetches the length, name of subprogram/function, number of arguments it has, the arguments in terms of entry numbers of tables, information which says whether it is a statement function, and link to definition if entry is a statement function, into variables lnth, token(1:1000), numarg, arg(1:20), defbit, and defent respectively. If the entry number given is a wrong one, then the logical variable errflg is returned as 'true'.

(e) Procedure endsfn: This routine puts an end marker in location one, at the end of all entries of the table.

(f) Procedure schsfn (lnth, var(1:10), found, entno): The input parameters to this routine are variables lnth, and var (1:10). The output parameters are variables found, and entno. The routine searches the table for the name of subprogram/function given in var(1:10). If it is successful in finding it in the table, then the logical variable found is returned as 'true' and the variable entno is returned with the value of entry number where found and a digit five appended at the end. If it is not found then logical variable found is returned as 'false'.

4-4.6 Routines dealing with statement number table

All the routines dealing with this statement number table use a common area labelled 'stmtno' which contains variables isfntb(1:3; 1:100), and istmpt. The table size is 100 and each entry has 3 locations. The variable istmpt always points to the next available entry in the table.

(a) Procedure inistn: This routine initialises variable istmpt to one.

(b) Procedure entstn (nmbr, entno, errflg): The input parameter to this routine is variable nmbr and the output parameters are variables entno and errflg. The routine enters the statement number inputted. The variable nmbr contains statement number. If statement number to be entered is a dummy one (-9999), then it is entered at the entry pointed by istmpt, and entno is returned with the value of entry number after appending digit 6 at the end.

If statement number to be entered is not a dummy one, then this number is searched in the table. If it is not present, then it is entered at the entry pointed by istmpt. The variable entno is returned with entry number to which a digit 6 is appended at the end. The digit 6 at the end indicates that this entry number is of statement number table. The logical variable errflg is returned as 'true' if there is no space to enter the given number.

(c) Procedure adjstn (entno, fmtflg, num, errflg): The input parameter to this routine are variables fmtflg, num, and entno. Output parameter is variable errflg. This routine enters the details of a statement number, when the entry number where statement number is stored is given. The entry number is value contained in variable entno after the last digit is truncated. The routine enters information regarding statement number, and link which are available in variables fmtflg, and num. Fmtflg will be 1 if statement number is a format statement number, in which case link points to the entry in comment and format table where corresponding format specification is stored. If fmtflg is zero then link points to the number of flow block started by this statement number. If the given entry number is a wrong one, then errflg is returned as 'true'.

(d) Procedure fchstn (entno, stno, fmtflg; link errflg): The input parameter to this routine is variable entno and output parameters are variables stno, fmtflg, link and errflg. Given the entry number, the routine fetches the statement number,

information which says whether it is a format statement number, and link to comment and format table or flowblock and places into variables stno, fmtflg, and link respectively. If the entry number given is a wrong one, then the logical variable errflg is returned as 'true'.

(e) Procedure endstn: This routine simply puts an end marker at the end of all entries of the table. The marker is put in first location of the entry.

(f) Procedure stnosz (itblsz): The variable itblsz is output parameter of this routine. This routine gives out the number of entries filled in statement number table at any time. This information is passed through variable itblsz.

(g) Procedure maxflb (iflb): Variable iflb is output paramter of this routine. This routine searches the statement number table and gets the information about the number of flow blocks used. This is obtained by seeing all non-format statement number entries which give information about flow blocks they are starting. The variable iflb contains this information about number of flow blocks. Usually this is called at the end of processing of a segment to know how many flow blocks it has used.

4-4.7 Routines dealing with segment header

All the routines dealing with segment header use a commong area labelled 'segment' which contains variable nsghdr(1:50).

(a) Procedure iniseg (nflbno): The input parameter to this routine is variable nflbno. This routine initialises first three locations of nsghdr(1:50). These locations contain information about type of segment, area where comments of beginning can be stored, and the first flow block being used.

(b) Procedure adjseg (class, length, token (1:1000), sbrfnc, exptyp, typcl, errflg): The input parameters to this routine are variables class, length, token (1:1000), sbrfnc, exptyp, and typcl. The output parameter is logical variable errflg.

The routine first saves the second and third locations of nsghdr(1:50), which are put at the end afterwards. Now value contained in variable sbrfnc, which specifies whether the segment is a subroutine, or function etc., is stored in first location of nsghdr(1:50). If exptyp is 1 then the function is explicitly declared and class of declaration given by typcl is stored in 3rd location of nsghdr(1:50). Exptyp is stored in second location of nsghdr(1:50). After these the length of name, and name of subprogram available in variables length, and token (1:1000) respectively are stored. The routine then gets the arguments, if any, and stores them as it is, The information about number of arguments precedes the arguments. A marker is also put at the end of all information stored in nsghdr (1:50).

(c) Procedure fchseg (nseg(1:50)): This routine simply copies the contents uptc end marker of nsghdr(1:50) into variable nseg(1:50) and the variable nseg(1:50) is passed out as output parameter.

4-5. FLOW, CODE, AND DECLARATION BLOCK ROUTINES

4-5.1 Routines dealing with flow blocks

All routines dealing with flow blocks use a common area labelled 'flowbl', which contains variable nflblk (1:30, 1:30). This means we are allowing thirty flow blocks. One need to change the common card in these routines, if one wants to have more flow blocks. Each flow block has 30 locations.

(a) Procedure iniflb (iflbno): There is no input parameter tor this routine but the output parameter is variable iflbno. The routine initialises variable iflbno to zero.

(b) Procedure getflb (iflbno, errflg): The input parameter to this routine is variable iflbno, and output parameters are variables iflbno, and errflg. This routine gives out the next flow block number. This is done by incrementing variable iflbno by one. And in the new flow blocks, in fact in all flow blocks, the first location points to the location of this flow block which is available for usage. So this routine initialises the first location to have a value of two. If there is no flow block which is available, then the logical variable errflg is returned as 'true'.

(c) Procedure entflb (type, stno, kdbno, iflbno): In this routine all the parameters are input parameters. The routine puts type, stno and kdbno in the flow block specified by variable iflbno in the locations two, three and four respectively. The variables type, stno, and kdbno specify the type of segment being processed, statement number which is starting this flow block, and the code block number where the corresponding statements of flow block are being stored respectively. The routine adjusts the first location of the flowblock to have a value of five.

(d) Procedure nxtflb (iflbno, nentry, errflg): The input parameters are variables iflbno, and nentry. The output parameter is logical variable errflg. This routine simply puts the value given by variable nentry in the available location of flow block specified by iflbno. If there is no available space in flow block, then a continuation mark is put in 30th location of flow block, and another flow block is obtained and the variable nentry stored. In any case the first location is updated so that it points to the next available location.

(e) Procedure endflb (iflbno): This routine puts an end marker at the end of all entries in the flow block specified by variable ilbno.

(f) Procedure chkflb (iflbno, wound): The input parameter to this routine is variable iflbno, and output parameter is logical variable wound. The routine just checks whether the

flowback specified by iflbno is wound or not. This check is done by means of checking end marker at the end of all entries. In case that the flowblock is found as wound, then logical variable wound is returned as 'true'.

(g) Procedure gtkdno (nocdbl, iflb): Given a flow block, this routine gets the corresponding code block where the statements of this block are stored. Variable iflb specifies for which flow block, we want information about its code block, and variable nocdbl contains information about the code block, which is returned as output parameter.

4-5.2 Routines dealing with code block

All routines which deal with code blocks use a common area labelled 'codebl' which contains variables kdblk(1:100,1:30), and kdbptr. The number of code blocks available are thirty and each code block has hundred locations. The variable kdbptr always points to the available location in the code block being processed.

(a) Procedure inicdb (kdbno): There is no input parameter for this routine but the output parameter is variable ikdbno which is initialised to zero by this routine.

(b) Procedure getcdb (kdbno): The variable kdbno is input and output parameter as well. The routine gets next code block and gives the number of this. This is done by incrementing kdbno by one. Also the variable kdbptr is initialised to one by this routine.

(c) Procedure entcdb (kdbno, class, length, token (1:1000), entno): All the parameters of this routine are input parameters. This routine enters a token or an entry number along with its class and length. If there is no space available to enter, then a continuation mark is put in the location pointed by kdbptr and another code block obtained and these stored. If the variable length has a value zero then class, length, and entno are stored in code block. Otherwise class, length, and token (1:length) are stored in the code block. The kdbptr, of course, is updated to point to next available location in the code block.

(d) Procedure endcdb (kdbno): This routine simply puts an end marker, at the end of all entries in the code block specified by variable kdbno.

(e) Procedure kdtkn (kdbno, class, length, token (1:1000)) The input parameter to this routine is kdbno and output parameters are variables class, length, and token(1:1000). This routine gets next available token from the code block specified by kdbno. The variable kdbptr will be pointing to location from where the information of token is available. If there is no token but a continuation mark is present, then this information is passed through variable class and control is returned. The routine, if a token is present, passes class, length and token present in code block through variables class, length, and token(1:1000). If length contains a value zero, then it indicates that token is in terms of entry number of some table.

(f) Procedure cheokd (kdend, kdbno): The input parameter to this routine is variable kdbno and output parameter is logical variable kdend. This routine checks to see if there are any more tokens in the code block specified by kdbno. This is known by checking for an end marker at the location pointed by kdbptr. If end marker is found then variable kdend is returned as 'true', indicating that there are no more tokens in code block.

4-5.3 Routines dealing with declarative block

All routines dealing with declarative block use a common area labelled 'declbl', which contains variables idclbl (1:200), and idclpt. The declarative block size is two hundred locations. The variable idclpt always points to the next available location in declarative block.

(a) Procedure inidcl: This routine initialises the variable idclpt to one.

(b) Procedure entdcl (class, length, token (1:1000), entno, errflg): The input parameters to this routine are variables class, length, token (1:1000), and entno. The output parameter is the logical variable errflg. If the variable length has a value zero, then this routine enters class, length, and entno in the available space. Otherwise class, length, and token (1:length) are entered in the available space. If there is no space available in the declarative block, then the logical variable errflg is returned with value 'true'.

(c) Procedure enddcl: This routine puts an end marker at the end of all entries in the declarative block.

(d) Procedure chendl (dclend): The logical variable dclend is output parameter of this routine. This routine checks whether there are any more tokens in declarative block. This is achieved by checking location of declarative block as pointed by idclpt for end marker. If it is found that there is end marker at location idclpt, then dclend is returned as 'true'.

(e) Procedure dcltkn (class, length, token (1:20)): The variables class, length and token (1:20) are all output parameters. This routine gets next token which is available alongwith its class and length in the declarative block. This information is available from location idclpt, Token got alongwith its class and length are passed out through variables token (1:20), class and length respectively. If the value of length is zero, then it indicates that token is in terms of entry number of some table.

CHAPTER 5

DETAILS OF STORAGE ROUTINES

In this chapter the routines dealing with input part i.e., which stores given FORTRAN program in the form of a flowgraph are described.

5-1. PROCEDURE PRSGMT (temp(1:72), iflbno, kdbno, errflg, endfle, doflg):

This routine processes a program segment. The input parameter is variable temp(1:72). The output parameters are variables iflbno, kdbno, errflg, endfle, and deflg.

The first statement of the segment being processed is already available in the form of tokens stored in variable koutlx(1:1000). After this statement a card is read and is available in variable temp(1:72).

The routine first initialises all the tables, blocks, stack, etc. It initialises segment header for main. Then it gets flow and code blocks for usage. The routine first checks whether this segment is a subroutine or function, in which case it modifies the segment header to contain the name and arguments of subroutine or function. A new statement in the form of tokens is then obtained.

The routine puts a 'continue' statement in the code block at the beginning. Now it processes all statements by invoking routine 'prstmt' until an 'end' statement comes. When an 'end' statement is seen, it checks whether previous

flow block is wound or not. If it is not wound then it winds up flow block. After this it checks for emptiness of 'DO' stack. If stack is not empty, then routine returns variable doflg as true. The logical variable errflg is returned as 'true' if there is any error in the last statement processed. The logical variable endfle will be returned as true if end of deck is encountered. The variables iflbno and kdbno return values of the last flow block and code block used by the segment.

5-2. PROCEDURE prstmt (class, length, token (1:1000), iflbno, kdbno, type, idumno, errflg)

The input parameters to this are variables class, length, token (1:1000), iflbno, kdbno, type, and idumno. The output parameter is logical variable errflg.

The first token of the statement being processed, along with its class, and length is available to the routine in the variables token(1:1000), class and length. The variables iflbno, and kdbno specify the present flow and code blocks. The variable type specifies the type of segment (main or subroutine, or function etc.) under processing. The variable idumno is useful when we want a dummy statement number while processing a DO statement.

If the statement being processed is a comment statement then the routine enters this statement in comment and format table, and corresponding entry number in the code block. A marker is then put in code block to delimit the statement.

If the statement is a statement function then name is entered in subprogram/function table alongwith its arguments. The statement is entered in the code block after the different tokens are entered into the tables. While checking for statement function, we may land up with assignment statements (without statement number) and so these assignment statements are also processed whenever found. They are entered in the code block. If it is a declarative statement, then routine 'prdecl' is called to process this statement.

If a statement with statement number comes, then the statement number is entered in statement number table. Then the routine checks for format statement. If it is found as format statement, then it is entered in the comment and format table and this entry number placed in the statement number table in the corresponding entry where statement number is stored. If it is not a format statement, then this routine invokes the routine 'prblst' to which information regarding statement number is also passed alongwith other details.

5-3. PROCEDURE prblst (stno, nent, class, length, token(1:1000), iflbno, kdbno, type, idumno, errflg)

The input parameters to this routine are variables stno, nent, class, length, token(1:1000), iflbno, kdbno, type and idumno. The output parameter is logical variable errflg.

If the statement under processing has statement number, then the variable stno will be nonzero and nent gives entry

number where it is entered in statement number table. A token apart from statement number is available in <u>token (1:1000)</u>. Variables <u>class</u> and a <u>length</u> give the class and length of token, Variables <u>iflbno</u> and <u>kdbno</u> specify the present flow and code blocks that are being used. Variable <u>type</u> tells the nature of segment and variable <u>idumno</u> is useful to get a dummy statement number, when a DO statement occurs.

If the statement has got statement number, then the routine checks whether the flow block is wound up as this statement starts new blocks. If flow block is not wound up, then the routine winds it up and gets fresh flow and code blocks. The entry in statement number table where this statement number is stored is adjusted to store information regarding flow block, being started by it. Information regarding statement number etc. is stored in flow block. The statement number is entered in the code block and further processing of the statement is done depending on the type of statement.

If the statement under process is a DO statement, then this routine invokes a routine called 'procdo' to process this statement. If the statement is a non-control statement, then it is checked for 'call' statement. If it is a 'call' statement then the name of the subroutine called, alongwith its arguments is entered into subprogram/function table, if it is not present in the table. If the name is already

present in table, then it is checked for matching of number of arguments etc. The statement is stored in code block also. The non-control statement other than 'call' statement is entered into the code block. After the non-control statement is entered into the code block, it is checked to see whether this statement ends the range of any DO statement(s). If the statement ends range of a DO statement, then the statements 'do parameter = do parameter + step' and 'IF (do parameter.LE. final value of do parameter) GO TO dummy statement number, which starts DO statement range' are entered into code block. The flow and code blocks are wound up and new flow and code blocks got. The successor of previous flow block is present one. In the new code block 'continue' is entered. The above procedure of checking end of range of DO is repeated until the statement is not an end of range of DO statement.

Logical IF statement is processed as follows. The statement upto the target of logical IF is entered into code block and then, the code block is wound and flow block is saved. New flow and code blocks are brought and information of present flow block being successor of saved flow block is entered. A procedure 'prtgt' is called to process the target of logical IF statement. At the end of processing of logical IF statement, similar to non-control statements, it is checked to see whether it ends range of DO statement(s).

Control statements other than logical IF, DO and END are processed by routine 'prctl', which will be called by this routine in case these statements occur.

5-4. <u>PROCEDURE procdo (class, length, token(1:1000), iflbno, kdbno, type, idumno, errflg)</u>

Except for variable <u>errflg</u>, which is an output parameter to this routine, the rest all are input parameters of this routine. The token after the statement number, if any, is available in <u>token(1:1000)</u>. The class and length of token are available in variables <u>class</u>, and <u>length</u>. Variables <u>type</u> spells out the type of segment under processing and variable <u>idumno</u> is used to get a dummy statement number.

This routine first gets all the parameters of DO statement and enters them in appropriate tables. If the step of DO parameter is not given, then it is taken as one. The statement 'DO parameter variable = initial value' is stored in the code block. Now a dummy statement number is got and entered into statement number table. The entry number of this statement number is entered as successor of the flow block. The present flow and code blocks are wound up and new flow and code blocks got. Information regarding the flow block is entered in statement number table entry where dummy statement number is entered. In the new code block the statement 'continue' is entered. Of course this statement will have dummy statement number. In the 'DO' stack, information of Do range statement number, its entry number

in statement number table, entry numbers of DO parameter variable, initial value, final value, step value is pushed.

5-5. PROCEDURE prctl (class, length, token(1:1000), iflbno, kdbno, errflg)

This routine processes control statements other than logical IF, DO and END. The input parameters to this routine are variables class, length, token(1:1000), iflbno, and kdbno. Output parameter is logical variable errflg.

The first token of the control statement being processed is available in token(1:1000). The class and length of the token are available in variables class and length. Variables iflbno and kdbno specify the flow and code blocks being used.

The arithmatic IF statement is processed as follows. First the expression of arithmatic IF is entered in the code block. Then the three statement numbers after the expression are got and entered into code block after entering them in statement number table. The entry numbers of these statement numbers are entered as successors of present flow block. The flow and code blocks are wound up.

Ordinary GO TO statement is entered into code block. The statement number involved is entered into statement number table and this entry is entered as successor of the present flow block. The flow and code blocks are then wound up. In case of assigned and computed GO TO statements the whole statement is entered into code block. The statement numbers involved in these statements are processed by calling a routine 'entnmb' which enters them in statement number

table [illegible] and the entries are entered into flow block as successors. The flow and code blocks are then wound.

In case of stop/return statement, it is entered into code block as it is. The successor of flow block is taken as zero, which indicates stop/return statement. In this case also the flow and code blocks are wound up.

5-6. <u>PROCEDURE prtgt (class, length, token(1:1000), nsvebl, kdbno, iflbno, type, errflg)</u>

This routine process the target statement of logical IF. The input parameters to this routine are variables <u>class</u>, <u>length</u>, <u>token(1:1000)</u>, <u>nsvebl</u>, <u>kdbno</u>, <u>iflbno</u>, and <u>type</u>. The output parameter is logical variable <u>errflg</u>.

The first token of the target statement is available in variable <u>token(1:1000)</u>. The class and length of the token are available in variables <u>class</u> and <u>length</u>. The flow block in which the logical IF statement excluding this target is stored is given by variable <u>nsvebl</u>. Variables <u>iflbno</u> and <u>kdbno</u> specify the present flow and code blocks being used. Variable <u>type</u> specifies the type of segment which is under processing.

If the target is found to be a logical IF, or DO or END statement then logical variable <u>errflg</u> is returned as 'true' and no further processing is done by the routine. Non-control statement as target statement is processed as follows. The non-control statement is entered in the code block. A new dummy statement number -9999 is entered in the statement number table and this statement number starts next flow block.

The entry number of this is put as successor of the present flow block and saved previous flow block. Saved as well as present flow block and code block are closed. New flow and code blocks are obtained. In the new code block statement 'continue' is entered.

If the target is a control statement other than logical IF, DO and END, then it is processed as follows. The statement is processed further by invoking routine 'pretl' which enters the statement in code block after doing necessary processing. A new dummy statemant number -9999 is entered in the statement number table. This statemant number is saved to start next flow block. The entry number of this statement number is put as successor of present flow block. Present flow block, saved flow block, and code block are then wound. New flow and code blocks are then obtained. In the new code block a 'continue' statement is entered.

5-7. <u>PROCEDURE prdecl (class, length, token(1:1000), errflg)</u>

This routine processes declarative statement. The first token of the statement is available to this routine along with its class and length. This information is given to the routine through the variables <u>class</u>, <u>length</u>, and <u>token (1:1000)</u> The logical variable <u>errflg</u> is returned as 'true' if there is any error in the statement.

External statement is processed as follows. It is entered into declarative block after entering all variables declared in this statement in the subprogram/function table. Common and equivalence statements are entered into the declarative block as it is i.e., without entering in terms of entry numbers of tables. Iñ case of common statement any information regarding dimensioned variables is extracted and entered in the dimensioned variable table. Data statement is stored in the declarative block in terms of entry numbers of v variables or constants. All other declarative statements are stored in declarative block after entering the simple and dimensioned variables, if any, in respective tables.

5-8. PROCEDURE entvar (class, length, token(1:1000), dclbit, ntypcl, errflg)

This routine enters variables of declarative statement into declarative block. The input parameters are variables class, length, token(1:1000), delbit, and ntypcl. The output parameters are class, length, token(1:1000) and errflg.

The name of the variable along with its class and length are available in variables token(1:1000), class and length respectively. The variable dclbit tells whether the variable to be entered is declared explicitly, in which case ntypcl tells mode of declaration.

This routine first determines whether the variable is a simple one or dimensioned variable. If it is simple variable, then it is entered into simple variable table and corresponding entry number entered in the declarative block. If it is a dimensioned variable, then the maximum arguments are obtained. The variable along with information regarding arguments is entered into dimensioned variable table and the corresponding entry number is entered in the declarative block. The routine returns next token after the variable. This information is passed out through the variables, class length, and token(1:1000).

5-9. PROCEDURE entid (class, length, token (1:1000), entno, kdbno, ndclbt, errflg)

This routine enters an identifier of any statement into declarative or code block depending on ndclbt being 1 or 0 respectively. The name of identifiers alongwith its class and length are available in variables token(1:1000), and class and length respectively. Variable kdbno specifies in which code block to enter. Variables entno, and errflg are output parameters.

The routine checks whether the identifier is a simple variable or not. If it is a simple variable then it is entered into simple variable table and corresponding entry number is entered in declarative or code block depending on value contained by ndclbt. The variable entno is returned with entry number. If it is not a simple variable, then

the name is searched in dimensioned variable table and subprogram/function table. If it is found in any one of them, then the entry number is stored in declarative or code block. If it is not found in them, then it is taken as a function name and the routine 'prefun' is called to process the function call. The routine returns token after identifier, along with its class and length through variables token(1:1000), class, and length respectively.

5-10. PROCEDURE prefun (nsvlnt, nsvar(1:10), class, length, token (1:1000), entno, kdbno, errflg):

The input parameters to this routine are variables nsvlnt, nsvar(1:10), class, length, token(1:1000), and kdbno. The output parameters are variables entno, and errflg.

The name of the function to be entered alongwith its length is available in nsvar(1:10). A token after this is available in token(1:1000). Variable kdbno tells us in which code block to enter.

The name of the function is first entered in the subprogram/function table and corresponding entry number is saved. This entry number is entered in the code block. Next the arguments of the function are got in terms of entry numbers of tables. Of course these arguments are entered in the code block. This information regarding the arguments of the function is entered in the entry number saved of subprogram/function table. This routine returns next token after the function call. This information is passed out through variables class, length, and token(1:1000).

5-11. PROCEDURE inidum(no)

This routine initialises the variable no to zero.

5-12. PROCEDURE getdum(no)

This routine gives next successive negative integer to the value contained in variable no. This means that variable no is decremented by one and returned.

5-13. PROCEDURE lntxpr (class, length, token(1:1000), kdbno, errflg)

The input parameters to this routine are variables class, length, token(1:1000), and kdbno. The output parameter is logical variable errflg. The routine enters expression of logical and arithmatic IF statements into code block specified by kdbno.

The variables class, length, and token(1:1000) give the class, length, and token after the left parenthesis of either logical IF or arithmetic IF statement. This routine just enters the expression of logical or arithmetic IF statement into code block in terms of entry numbers of tables. It enters upto right parentheses of the expression and returns token after that through variable token(1:1000).

5-14. PROCEDURE dolxi (temp(1:72), endfle)

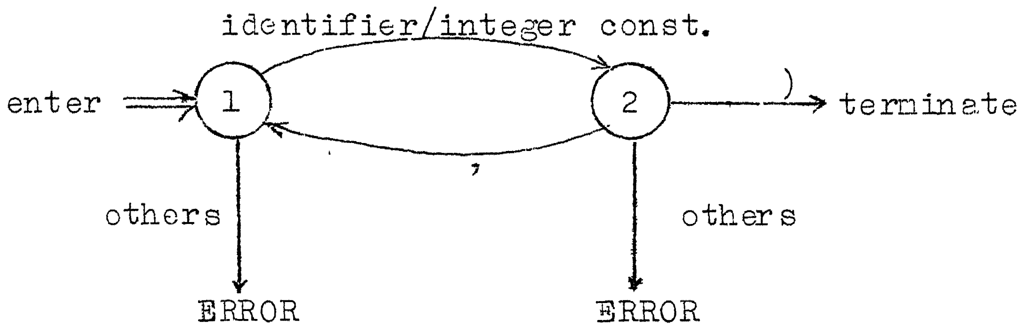
The input parameter to this routine is variable temp(1:72). The output parameters are variables temp(1:72) and endfle. A card is already read and available in temp (1:72). This routine collects a statement, lexical analyses and stores tokens in appropriate area. A card after the

statement lexical analysed is already read and is passed out through variable temp(1:72). The logical variable endfle is returned as 'true' if the card after statement lexical analysed indicates end of deck.

5-15. PROCEDURE getarg (class, length, token(1:1000), numarg, arg(1:5), jdclbl, errflg)

This routine gets arguments of dimensioned variables present in declarative statements. The input parameters are variables class, length, token(1:1000), and jdclbl. The output parameters are variables numarg, arg(1:5), and errflg.

The token after the left parentheses of dimensioned variable is available in token(1:1000). Finite state machine technique is used to get the arguments. The state diagram of machine is as follows.



So using the above finite state machine, this routine gets arguments one by one. They are entered in appropriate tables and entry numbers are entered into declarative block if variable jdclbl has value 1. At the end of execution of routine variable arg(1:5) contains the arguments of dimensioned variable in terms of entry

5-17. PROCEDURE inistk

This routine uses a common area labelled 'ptr' which contains variable istkpt. This variable istkpt is initialised to zero by this routine.

5-18. PROCEDURE pshstk (dornge, entno, parnt, fnent, stpent, errflg)

The input parameters to this routine are variables dornge, entno, parnt, fnent, and stpent which give information about a DO statement. The output paremeter is logical variable errflg.

This routine uses two common areas labelled 'ptr' and 'stack'. The first one contains variable istkpt and the second one contains variables idostn(1:20), jent(1:20), ipar(1:20), ifin(1:20), and istep(1:20). This means the size of stack is twenty.

If there is space in the stack, then variable istkpt is incremented by one and the variables dornge, entno, parnt, fnent, and stpent are pushed into the stack. The variable istkpt always points to the most recently entered entries.

5-19. PROCEDURE chkstk (empty)

This routine uses a common area labelled 'ptr' which contains variable istkpt. The output parameter is variable empty. The routine checks whether the stack is empty or not. This is done by means of checking value of istkpt. If istkpt contains zero, then the stack is empty and so

logical variable empty is returned as 'true'. If istkpt contains nonzero, then stack is not empty and so logical variable empty is returned as 'false'.

5-20. PROCEDURE chendo (stno, dument, parnt, fnent, stpent, doend, errflg)

The input paremeter to this routine is variable stno and the rest are output parameters. The routine makes use of two common areas labelled 'ptr' and 'stack'. The first one contains variable istkpt and the second one contains variables idostn(1:20), jent(1:20), ipar(1:20), ifln(1:20), and istep(1:20).

This routine checks to see whether the statement number given through variable stno ends the range of a DO statement. For this the top element of stack pointed by istkpt is checked to see if idostn (istkpt) and stno have same values. If they are same then the routine returns all the information about DO statement apart from returning logical variable doend as 'true'. The variable istkpt is decremented by one so that it points to next entry in DO stack. The logical variable doend is returned as 'false' if given statement number does not end range of a DO statement.

CHAPTER 6

DETAILS OF RETRIEVAL ROUTINES

In this chapter the components of outputpart, which prints a FORTRAN program from the information present in flowgraph, are described. In printing out statements a buffer of 72 locations is used. Characters of the statement are entered into buffer and when a line is complete, it is printed out. In Section 1 we describe some of the routines which are used by other parts. In Section 2, routines which deal with printing of segment header are described. Routines which deal with printing of declarative statements, code blocks and format statements are described in Sections 3, 4 and 5 respectively.

6-1. SUPPORTING ROUTINES

We describe one by one the different routines used by other sections.

6-1.1 Procedure entbuf (khar, j)

Both variables khar and j are input parameters. This routine uses a common area labelled 'outbuf' which contains variable motbf(1:72) into which a line is entered before printing it out.

The variable J points to location upto which the buffer motbf(1:72) is occupied. The routine enters khar in the next location J+1 of buffer. If the buffer is full already,

then the routine prints out the contents of buffer and puts blanks in first 5 locations of buffer, a continuation mark in 6th column, and the khar in 7th column and returns control.

6-1.2 Procedure inibuf(j)

This routine just initialises the variable J to zero. The variable J points to location upto which buffer is full.

6-1.3 Procedure prtbuf

This routine uses a common area labelled 'outbuf' which contains variable motbf(1:72). This routine prints out the contents of buffer motbf(1:72). It will not print out the contents if the buffer contains just a 'CONTINUE' statement.

6-1.4 Procedure bufcmf (entno, j)

The input parameters to this routine are variables entno and J. The variable entno gives entry number ofthe comment and format table, which has to be entered into buffer. The routine fetches the entry of comment and format table as pointed by variable entno. Then it enters the token fetched in the buffer.

6-1.5 Procedure bufcns (entno, j)

Both variables entno and j are input parameters to this routine. Variable entno gives entry number of constant table, from which the token or constant is fetched by the routine. If the constant obtained is a real or hollerith one, then it is entered into the buffer as they will be in character form. If the obtained constant is an integer constant, then it is seperated into individual digits and entered into the buffer in character form.

6-1.6 Procedure bufdmn (entno, j)

Both variables entno and j are input parameters to this routine. The routine fetches an entry, as pointed by variable entno, from the dimensioned variable table. The routine then enters the name of the dimensioned variable in the buffer.

6-1.7 Procedure bufsmp (entno, j)

Both variables entno and j are input parameters to this routine. This routine fetches an entry, as pointed by variable entno, from the simple variable table. The routine then enters the name of simple variable in the buffer.

6-1.8 Procedure bufsfn (entno, j)

The input parameters to this routine are variables entno and j. This routine fetches an entry, as pointed by variable entno from the subprogram/function table. The routine then enters the name of subprogram/function in the buffer.

6-1.9 Procedure bufstn (entno, j)

Both variables entno and j are input parameters to this routine. This routine fetches an entry, as pointed by variable entno, from the statement number table. The statement number obtained is separated into individual digits and stored in the buffer in character form.

6-1.10 Procedure sepcns (nmbr, ln, dig(1:20))

The input parameter to this routine is variable nmbr. The output parameters are varia bles ln and dig(1:20). The routine separates the positive number, given by variable nmbr, into individual digits and stores them in character form in the variable dig(1:20). The variable ln tells how many digits the given number has.

6-2. ROUTINES DEALING WITH PRINTING OF SEGMENT HEADER

There is only one routine in this part to be explained. It is described as follows:

6-2.1 Procedure prtseg (errflg)

There is no input parameter to this routine but the output parameter is logical variable errflg.

This routine first fetches the contents of the segment header. If it is found that the segment is a main one then control is returned. If it is found that the segment is a 'subroutine' then the routine first enters 'SUBROUTINE' in the buffer. The name and its arguments, if any, are entered latter in the buffer. After this the contents of buffer are printed out.

If the segment is found as 'function' then it checks whether the function is explicitly declared, in which case corresponding declaration (integer, real, etc.) is entered first in the buffer. Then 'function' is entered. After this the name of function and its arguments are entered in the buffer. We know that arguments are present in the segment

header as it is. At the end the contents of buffers are printed out. If the segment is 'black data' then it is entered in the buffer and printed out. Locations in the buffer, other than what we entered using segment header, are filled with blanks before printing out contents of buffer.

6-3. ROUTINES DEALING WITH PRINTING OF DECLARATIVE STATEMENTS

There is only one routine in this part to be explained. It is described as follows.

6-3.1 Procedure psctdcl (errflg)

There is no input parameter to this routine but the output parameter is logical variable errflg.

This routine prints out all declarative statements present in the declarative block. Statements in the declarative block are delimited by markers. Each statement is printed as follows.

A token of statement is processed as follows. If length of token is non-zero then whatever present in token (1:length) is entered in the buffer. If length is found to be zero, then we know that token is in terms of entry number of either simple or dimensioned variable table as only these variables will be present in declarative statements. If entry number is of simple variable table, then that entry is fetched from table and name of variable entered in buffer. If entry number is of dimensioned variable table, then that entry is fetched from that table. First the name of variable is

entered in buffer. Then using argument entry numbers, the arguments are fetched from constant table, separated into individual digits and entered into buffer. Of course the arguments are separated by commas and enclosed in parenthesis in the buffer. In this way all tokens of a declarative statement are processed and entered in buffer. At the end, the statement is printed out through buffer after entering blanks at appropriate places in buffer.

In the above said way all statements present in declarative block are printed out and control returned.

6-4. ROUTINES DEALING WITH PRINTING OF STATEMENTS IN CODE BLOCK

The different routines are described as follows.

6-4.1 Procedure prntcd (errflg)

There is no input parameter to this routine but the output parameter is logical variable errflg. This routine prints out contents of all code blocks.

The routine first gets the number of code blocks whose contents are to be printed out. Then it processes all code blocks. Statements in a code block will be delimited by end markers. A statement in a code block is processed as follows.

The first token of a statement is brought. It is seen if that token is a statement number entry. If it is found that statement has statement number, then it is fetched from statement number table, separated into individual digits and entered into the buffer in locations 1 to 5. The token after this statement number is brought and is processed as

follows.

If the length of token is found as non-zero, then token (1:length) is entered in the buffer. If the length is found as zero, then it indicates that the token is in terms of entry number of table. To which table the entry belongs to is known by examining the last digit of token(1). Then appropriate routine is invoked to fetch that entry and is entered into the buffer. When an end marker is seen while getting a token, then it is seen whether the statement being processed is logical IF statement. In this case of logical IF statement, it is checked to see if target is entered in the buffer. This is achieved by examining last token for right parenthesis. For this purpose variable klprev keeps track of class of last token entered in buffer. If target of logical IF is not entered in buffer, then routine 'prlgif' is called to do the same. The statement present in the buffer is then printed out after entering blanks at appropriate places. While processing a statement if a continuation mark is seen in the code block, then next code block is obtained and processing continued. In this way all statements in a code block are printed out. When statements in a code block are over, then next code block is obtained and processed in same manner as above. Thus this routine prints out contents of all code blocks.

6-4.2 Procedure prlgif (kdbno, J, errflg, class, length, token (1:1000))

Input parameters to this routine are variables kdbno and j. Output parameters are variables class, length, errflg and token (1:1000). This routine prints out target of a logical IF statement, whose expression is in code block specified by kdbno, and target in the next code block. Variable j tells upto what location the buffer is full.

The routine first checks to see that code block containing expression of logical IF statement has no other tokens in it. Then it gets next code block which has the target of logical IF. Tokens of this target statement are brought one by one and processed as follows.

If the length of the token is non-zero, then the token (1:length) is entered into the buffer as it is. If it is found that length of the token is zero, then it indicates that token is in terms of entry number of some table. To which table the entry belongs is detected by checking the last digit of token(1). Then appropriate routine is invoked to fetch that entry and is entered into the buffer.

The above procedure is repeated until all tokens are processed i.e., until an end marker is seen. Then the routine returns control back.

6-4.3 Procedure cdblsz (nocdbl)

The variable nocdbl is an output parameter of this routine. The routine finds the number of code blocks used by the segment. First this routine calls procedure called 'maxflb' to get to know how many number of flow blocks are used by segment. Then it calls routine 'gtkdno' to know the code block corresponding to last flow block used. This gives the number of code blocks used and is passed out through variable nocdbl.

6-5. ROUTINES DEALING WITH FORMAT STATEMENTS PRINTING

There is only one routine to be described and is explained below.

6-5.1 Procedure prtfmt (errflg)

The logical variable errflg is an output parameter of this routine, which prints out all format statements.

The routine first gets the number of entries present in the statement number table. Then each entry is fetched from the statement number table and examined to see if statement number is of a format statement. If it is found that it is a format statement number, then entry number in comment and format table where actual FORMAT statement is available is got. This entry is fetched and FORMAT statement is entered in the buffer and printed out after putting blanks in appropriate places. The above procedure is repeated until all the entries in statement number are examined.

CHAPTER 7

FUTURE WORK AND CONCLUSIONS

A package, which can read a 80 column card deck of FORTRAN program segment and store it in the form of flow-graph and which can convert the flowgraph back into FORTRAN program, has been designed and implemented.

7-1. FURTHER WORK THAT CAN BE DONE

This project can be extended to achieve some program manipulations. Some of the things which can be done are as follows:

Optimization, in terms of execution speed improvements, of the given FORTRAN program can be achieved since this package stores the given FORTRAN program in a convenient form for such manipulation. One more thing that can be done is compilation of the given FORTRAN program. Data flow analysis of the given FORTRAN program also can be done using this package.

7-2. CONCLUSIONS

The present package is thus useful in achieving program manipulations. In its present form this package can deal FORTRAN program segments which can split into less than thirty blocks. If one wants to extend this to deal big programs, he just need to change the labelled COMMON statements in necessary modules. For example, if one wants to increase the size of comment and format table, what all he

needs to do is to just change the labelled COMMON statements present in routines of the module dealing with COMMENT and FORMAT table. One limitation of this package is it does not allow identifiers which start with any of the reserved word.

BIBLIOGRAPHY

1. Allen, F.E., and Cocke, J., (1976), 'A Program Data Flow Analysis Procedure', CACM, Vol. 19, No. 3, March 1976, pp. 137-147.

2. Baker, B.S., (1977), 'An Algorithm for Structuring Flowgraphs', Jour. of ACM, Vol. 24, No. 1, Jan. 1977, pp. 98-120.

3. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R., 'Structured Programming', London : Academic Press.

4. IBM 7040/7044 Operating System (16/32K); Programmers Guide, Form C28-6318-6.

5. Johnson, W.L., Porter, J.H., Ackley, S.I., and Ross, D.T., (1968), 'Automatic Generation of Efficient Lexical Processors Using Finite State Techniques', CACM, Vol. 11, No. 12, pp. 805-813.

6. Lecht, C.P., 'The Programmer's FORTRAN II and IV : A Complete Reference,' Mc-Graw-Hill Book Company.

7. Lowry, E.S., and Medlock, C.W., (1969), 'Object Code Optimisation', CACM, Vol. 12, No. 1, Jan. 1969, pp. 13-22.

8. Parnas, D.L., (1972), 'On the Criteria to be Used in Decomposing Systems into Modules', CACM, Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.

9. Schaefer, M., 'A Mathematical Theory of Global Program Optimization', Prentice-Hall, Inc., London.

10. Schneck, I.B., and Angel, E., (1973), 'A FORTRAN to FORTRAN Optimizing Compiler', Computer Journal, Vol. 16, No. 4, 1973, pp. 322-330.

11. Schantz, I.W., German, R.A., Mitchell, J.G., Shirley, R.S.K., and Zarnke, C.R., (1967), 'WATFOR - The University of Waterloo FORTRAN IV Compiler', CACM, Vol. 10, No. 1, Jan. 1967, pp. 41-45.

12. Standish, T.A., Harriman, D.C., Kibler, D.F., and Neighbors, J.M., 'The Irvine Program Transformation Catalogue'.

13. Stevens, W.P., Myers, G.J., and Constantine, L.L., (1974), 'Structured Design', IBM Systems Journal, Vol. 15, No. 2, pp. 115-139.

14. Wirth, N., 'Systematic Programming : An Introduction', Prentice-Hall, Inc., New Jersey.

APPENDIX A

Token	Class
Key words	-1
Identifier	0
Integer Number	1
Real Number	2
*	3
Unary + / -	17
Binary + / -	4
/	5
=	6
(	7
)	8
,	9
. $	10
'	11
**	12
Comment	13
Statement Number	14
Hollerith	16

Reserved word	Class
Logical IF	-70
Arithmetic IF	-80
END	-90
STOP/RETURN	-82
CONTINUE	-60
WRITE	-60
FORMAT	-40
GO TO	-81
READ	-60
PRINT	-60
PUNCH	-60
PAUSE	-62
DATA	-30
CALL	-61
REAL	-21
INTEGER	-20
LOGICAL	-22
COMPLEX	-23
COMMON	-28
EQUIVALENCE	-25
FUNCTION	-11
SUBROUTINE	-10
DIMENSION	-26
DOUBLE PRECISION	-24
REWIND	-60
ENDFILE	-60
BACK SPACE	-60
ASSIGN	-60
DO	-50
TO	-99
EXTERNAL	-29
BLOCKDATA	-12

APPENDIX B

Character(s)	Major Class	Individual Class
blank	1	1
Letters A-D,F,G,I-Z	2	23-26,28,29,31-48
Letter H	3	30
Letter E	4	27
Digits 0-9	5	13-22
+ or -	6	3,4
*	7	2
.	8	12
Delimiters /,=,(,),,,$,'	9	5-11

APPENDIX C

The following are the routines and are described in pseudo-computer language [Chapter 2].

1. Procedure clstmt (stmt (1:700), temp(1:72), endfle, lnstmt, error);

```
  begin comment:  This routine collects a statement from
                  input; Prior to this a card has already
                  been read into 'temp';
      error ← 0; endfle ← 'false';
      if (temp(1) ≠ comment) then
                  begin comment: check for col. 6 of previous
                                 card read;
                        if (temp(6)≠blank ∧ temp(6)≠0) then
                           begin comment: error-col. 6 ignored.
                                 temp(6) ← blank; error ← error+1;
                           end
                        fi
                  end
      fi
      comment: now move previous card read, into 'stmt';
      for j = 1 to 72 do
          stmt(j) ← temp(j);
      od
      lnstmt ← 72; chkflg ← 'false';
      while (¬chkflg) do
          read a card;
          if (temp(1)≠comment ∧ stmt(1)≠comment) then
             begin comment: check for end of file marker;
                   if (temp(1)=slash ∧ temp(2)=slash) then
                      begin endfle ← 'true';
                            chkflg ← 'true';
                      end
                   else comment: check for continuation
                                 card;
                        if (temp(6)=blank ∨ temp(6)=0) then
                           begin chkflg ← 'true'; end
                        else comment:we got cont.card;
                             if(lnstmt ≥ 666) then
                                begin error; [illegible]
                                      chkflg ← 'true'
                                end
                             else comment: check for
                                  blanks in col.1-5;
                                  for m = 1 to 5 do
                                    if(temp(m)≠blank)
                                       then error;
                                    fi
                                  od
```

```
                                        for k=7 to 72 do
                                           lnstmt ← lnstmt+1;
                                           stmt(lnstmt) ←
                                                  temp(k);
                                        od
                                    fi
                                fi
                           fi
                       end
                   else chkflg ←'true';
               fi
           od
           comment: now we put a marker at the end of stmt.
           stmt(lnstmt+1) ← marker;
       end
```

2. Procedure nxtchr (stmt(1:700),i);

```
  begin comment: This gets next character present at i+1th
                 position in stmt. along with its classes
        if (klch≠klblank) then
          begin kprev←klch; kprev2←klass2; end
        fi
        i ← i+1; char ←stmt(i);
        chrcod(char, kodch, klch, klass2);
  end
```

3. Procedure chrcod (char,kodch,klch,klass2);

```
  begin comment: Given a character, this gets its code
                 and different classes;
        ncnst ← 2 ↑ 30; kodch ←char/ncnst;
        if (char < 0) then kodch ← 32-kodch; fi
        klch ←istcls(kodch+1); klass2 ←indcls(kodch+1);
  end
```

4. Procedure fsmtbl (state, klch, nxtact, nstate);

```
  begin: comment: Given present state, and class of character
         got, this gives next state and action to be taken;
         ntemp ←mchtbl (state, klch);
         nxtact← ntemp/100;
         nstate ←ntemp - nxtact * 100;
  end
```

5. Procedure newoul;

```
  begin  kount ← 1; end
```

6. Procedure newtkn;

```
  begin klsptr ←kount;
        koutlx(kount) ←-1; koutlx(kount+1) ←-1;
        kount ← kount+2;
  end
```

```
7.    Procedure addchr(khar);
  begin koutlx(kount) <-khar;
        kount <- kount+1;
  end

8.    Procedure defcls(klas);
  begin koutlx(klsptr) <- klas; end
9.    Procedure  endtkn;
  begin kountlx(klsptr+1) <- kount - klsptr-2; end

10.   Procedure endount;
  begin koutlx(kount) <-marker; end

11.   Procedure ensint(n);
  begin comment: This constructs an integer from individual
                 digits of token, and stores it;
        noofdg <-kount - klsptr-2; n <-0; ncn <- 2  ↑ 30;
        for i = 1 to noofdg do
            nptr <-klsptr + i + 1;
            n <-n * 10 + koutlx(nptr)/ncn;
            kount <- kount-1;
        od
        koutlx (kount) <- n;
        kount <- kount+1;
  end

12.   Procedure newlxi;
  begin kount <- 1; end

13,   Procedure cheolx (endlxi);
  begin comment: This one checks whether tokens of stmt. are over;
        endlxi <- 'false';
        if (kountlx (kount) = marker) then endlxi <-'true'; fi
  end

14.   Procedure fchtkn (class, length, token(1:1000));
  begin comment: This fetches a token stored in 'koutlx';
        class <-koutlx (kount); length <-koutlx(kount+1);
        kount <- kount+2;
        for j = 1 to length do
            token (j) <-koutlx(kount);
            kount <- kount+1;
        od
  end

15.   Procedure restbl;
  begin comment: This routine constructs reserved word table;
        reads all reserved words along with details;
  end
```

```
16.      Procedure strtwd (char, resflg);
   begin comment: This routine checks first letter of
         identifier with possible reserved word;
         resflg ← 'false'; nptr ← iroot;
         while (nptr≠0) do
           if (kode(char) = kode(mchar (nptr)))then
                begin resflg ← 'true'; nptr ← msucc(nptr);end
             else if (kode(char) > kode(mchar(nptr)) then
                                           nptr ← idnalt(nptr);
                                       else nptr ← iupalt(nptr);
                 fi
           fi
         od
   end

17.      Function procedure kode(m)
   begin comment: This gives internal code of m;
         nconst ← 2 ↑30; kode ← m/nconst;
         if (kode < o) then kode ← 32-kode; fi
   end

18.      Procedure oknxch (char, signal, next, class);
   begin comment: This routine searches character other than
         first of an id with a corresponding char in res. word;
         signal ← -1
         if (khar(nptr) = endmrk) then
                   begin signal ← 2; next ← isux(nptr)/100;
                         class ← -(isux(nptr) - next * 100);
                         nptr ← malt (nptr);
                   end
             else if (char = khar (nptr)) then
                          begin signal ← 1; nptr ← isux(nptr); end
                     else nptr ← malt(nptr);
                          while (nptr≠ 0) do
                            if (char=khar(nptr)) then
                                  begin nptr ← isux(nptr);
                                        signal ← 1; return;
                                  end
                              else nptr ← malt(nptr);
                            fi
                          od
                          signal ← 0;
                 fi
         fi
    end
```

```
19.        Procedure lexcal (stmt(1:700),i,lnstmt, error);
   begin comment: This one outputs tokens of stmt.(1:700)
                  Already a non-blank character, its code,
                  classes have been got.
        newoul;
        comment: first we check for comment stmt.
        if (i=1 ∧ char = count) then
                  begin newtkn; defcls (cmtcls);
                        for i = 1 to 72 do addchr (stmt(i)); od
                        endtkn; endoul; return
                  end
           else comment: we check for stmt. no. first;
                if (i ≤ 5) then
                      begin newtkn; defcls (stmtcl);
                            Repeat
                             if (stmt(i)εdigit) then addchr(stmt(i));
                                   else if(stmt(i)≠blank)then
                                           error;
                                        fi
                             fi
                             i ← i+1;
                             until i > 5
                             cnstnt (mn); endtkn;
                             nxtchr(stmt(1:700),i);
                      end
                fi
           outloop: newtkn; state ←1;
              loop: fsmtbl (state, klch, nxtact, nstate);
                    case class of
                      blank: state ← nstate;
                      class          nxtchr(stmt(1:700),i);
                                     go to loop;
                      char.: addchr(char);
                      class  state ← nstate;
                             nxtchr(stmt(1:700),i);
                             go to loop;
                      delim.:comment: we got delimiter other
                      class           than + / - , *
                             addchr(char); defcls(klass2);
                             endtkn;
                             if(klass2=keol) then go to last;
                                else nxtchr(stmt(1:700),i);
                                     go to outloop;
                             fi
                      +/-   :comment: we got +/- check for
                      class           unary or binary;
                             addchr(char);
                             if(kprev2=kequal ∨ kprev2=klpar)
                                       then defcls(unplmi);
                                       else defcls (bnplmi);
                             fi
                             endtkn;nxtchr(stmt(1:700),i);
                             go to outloop;
```

```
 *      : comment: we got *. Check
 class              for another *
          addchr(char);
          Repeat nxtchr(stmt(1:700),i);
            until (char≠blank)
          if (char=star) then
                begin  addchr(char);
                       defcls(dblstr);
                       nxtchr(stmt(1:700),i)
                end
             else defcls (kstar);
          fi
          endtkn;
          go to outloop;
 idcls:  comment: we got an identifier;
         defcls(idcls); endtkn;
         go to outloop;
 intcls: comment: we got an integer no.;
         defcls(intcls); cnsint(num);
         endtkn;
         go to outloop;
 realcl: comment: we got a real no.;
         defcls(realcl); endtkn;
         go to outloop;
 kywdcl: Comment: we got a keyword;
         addchr(char); defcls(kywdcl);
         endtkn; nxtchr(stmt(1:700),i);
         go to outloop;
 holrcl: comment: we got a hollerith const.;
         cmsint(number);addchr(char);
         defcls(holrcl);
         for k = 1 to number do
             i ← i+1;
             addchr (stmt(i));
         od
         endtkn; nxtchr(stmt(i));
         go to outloop;
first   :comment: we got first letter of
letter             identifier/res. word;
class    addchr(char);
         strtwd(char, resflg);
         if(resflg) then state ← nstate;
                    else state ← 2;
         fi
         nxtchr(stmt(1:700),i);
         go to loop;
```

```
other : comment: we got another char.check
char              for match in res. word;
class   oknxch(char, signal,next,class);
        if (signal=0) then
                      begin addchr(char);
                            state ← 2;
                            nxtchr(stmt(1:700),i);
                            go to loop;
                      end
              else if (signal=1) then
                            begin addchr(char);
                                  state ←nstate;
                                  nxtchr(stmt(1:
                                    700),i);
                                  go to loop;
                            end
                      else case class of
                            docls: comment:we got
                                               do;
                                  if(klch≠digcls)
                                    then
                                    state  nstate;
                                    go to loop;
                                  fi
                                  defcls(class);
                                  endtkn;
                                  put stmt.no.
                                  after DO as
                                  taken in
                                  output;
                                  go to outloop;
                          endcl: comment: we got
                                               'END'
                                 if(klass2=keol)
                                    then
                                    defclas(class);
                                    endtkn;
                                    go to outloop;
                                    else
                                    state←nstate;
                                    go to loop;
                                 fi
```

```
                                        reswd: comment: we got
                                        class           res.
                                                        wd other
                                                        than do,
                                                        end,
                                                        format;
                                               defcls(class);
                                               endtkn;
                                               go to outloop;
                                        format:comment: we got
                                        class           format;
                                               defcls(class);
                                               for m=2 to lnstmt
                                                     do
                                                addchr(stmt(m));
                                               od
                                               endtkn;
                                               nxtchr(stmt(1:
                                                700),lnstmt);
                                               go to outloop;
                                        end
                                   fi
                          fi
                    complete: comment: we got complete res.word/
                    wd.class           identifier
                             oknxch(char,signal,next,class);
                             if(signal≠1) then defcls(idcls);
                                               endtkn;
                                               go to outloop;
                                else if (class=fmtcls) then
                                         begin defcls(class);
                                               for m=i to lnstmt
                                                   do
                                                    addchr(stmt(m));
                                               od
                                               endtkn;
                                               nxtchr(stmt(1:
                                               700),lnstmt);
                                               go to outloop;
                                         end
                                         else defcls(class):
                                              endtkn;
                                              go to outloop;
                                     fi
                             fi
                   error : error; return
                   class
                 end
```

```
          last: newlxi; fchtkn(class,length,token(1:1000);
                if (class = ifcls) then
                    begin comment: we separate logical IF
                                   and arith. IF
                              i←token(1); ii←token(1);
                              fchtkn(class,length,token(1:1000));
                              iparct ←1;
                              while (iparct≠0) do
                                fchtkn(class,length,token(1:1000))
                                if(class=lparcl)then
                                  iparct ←iparct+1;  . i
                                fi
                                if (class=rparcl) then
                                  iparct ←iparct-1;
                                fi
                             od
                             fchtkn(class,length,token(1:1000));
                             newoul; newtkn; addchr(i);
                             addchr(ii);
                             if(class=intcls) then defcls(arifcl);
                                                 else defcls(lgifcl);
                             fi
                             endtkn;
                    end
                  else comment: we seperate function sub.
                                program such as integer
                                function etc.;
                      if(class ε declcl) then
                          begin for i = 1 to length do
                                        itemp(i) ← token(i);
                                  od
                                  nlnth ← length;
                                  fchtkn(class,length,token
                                          1:1000));
                                  if (class=fncl) then
                                      begin newoul;newtkn;
                                              k← class-100
                                              defcls(k);
                                              for j=1 to nlnth
                                              do
                                              addchr(itemp(j));
                                              od
                                              endtkn;
                                      end
                                  fi
                          end
                      fi
                fi
      fi
end
```

```
20.        Procedure inicmt;
   begin kmtptr ← 1; end

21.        Procedure entlne (class, length, token(1:1000),
                                  entno, errflg);
   begin comment: This routine enters a comment or format
                  statement into the table;
         if (kmtptr ≥ kmtsze) then error; return; fi
         entno ← kmtptr * 10 + 1;
         kmtbl(1, kmtptr) ← class;
         kmtbl (2, kmtptr) ← length - iniptr + 1;
         k ← iniptr;
         while (k ≤ length) do
           strptr ← 3;
           klim ← min(length,k,71);
           while (k ≤ klim) do
             kmtbl (strptr, kmtptr) ← token(k);
             strptr ← strptr+1; k ← k+1;
           od
           if (k ≤ length) then kmtbl(75,kmtptr) ← kntmrk;
                           else kmtbl(75,kmtptr) ← nocnt;
           fi
           kmtptr ← kmtptr+1;
         od
   end

22.        Procedure fchlne (class, length, token(1:1000),entno,
                                         errflag);
   begin comment: This fetches an entry of table, if entry
                  number is given;
         if (entno ≥ kmtsze) then error; return; fi
         nptr ← entno;
         class ← kmtbl(1,nptr); length ← kmtbl(2, nptr);
         k ← 1;
         while (k ≤ length) do
           strptr ← 3; klim ← min(length, k+71);
           while (k ≤ klim) do
             token(k) ← kmtbl(strptr, nptr);
             strptr ← strptr+1; k ← k+1;
           od
           if(k ≤ length) then
             if(kmtbl(75,nptr)≠kntmrk)then error; return; fi
             else
             if(kmtbl(75,nptr)≠nocnt) then error; return; fi
           fi
           nptr ← nptr + 1;
         od
   end
```

```
23.      Procedure endcmt;
   begin kmtbl(1,kmtptr) ← endmrk); end

24.      Procedure inicns;
   begin knsptr ← 1; end

25.      Procedure entcns(class,length,token(1:1000),entno,
                                    errflg);
   begin comment: This enters a constant into the table;
         if (knsptr ≥ knsze) then error; return fi
         entno ← knsptr * 10+2;
         knstbl(1,knsptr) ← class;
         knstbl(2,knsptr) ← length;
         k ← 1;
         while (k ≤ length) do
           strptr ← 3; klim ← min(k+16, length);
           while (k ≤ klim) do
             knstbl(strptr, knsptr) ← token(k);
             k ← k+1; strptr ← strptr+1;
           od
           if (k ≤ length) then knstbl (20,knsptr) ← kntmrk;
                              else knstbl(20,knsptr) ← nocnt;
           fi
           knsptr ← knsptr+1;
         od
   end

26.      Procedure fchcns (class,length, token(1:1000),entno,
                                        errflg);
   begin comment: This routine fetches an entry from table, if
                  entry number is given;
         if (entno ≥ knsze) then error; return fi
         class ← knstbl (1,entno);
         length ← knstbl(2,entno);
         nptr ← entno; k ← 1;
         while (k ≤ length) do
           strptr ← 3; klim ← min(k+16, length);
           while (k ≤ klim) do
             token(k) ← knstbl(strptr, nptr);
             k ← k+1; strptr ← strptr+1;
           od
           if (k ≤ length) then
               if (knstbl(20,nptr)≠kntmrk)then error; return fi
               else if (knstbl (20,nptr)≠nocnt) then error;
                                                    return
                   fi
           fi
           nptr ← nptr+1;
         od
   end

27.     Procedure endcns;
   begin knstbl(1,knsptr) ← marker; end
```

28. Procedure inidmn;

```
begin dmnptr ← 1; end
```

29. Procedure entdmn (lnth, token(1:1000),exptyp,typcl, entno, errflg);

```
begin comment: This enters a dimensioned variable along
               with some deails into the table;
      if (lnth > 6) then error; return; fi
      if (dmnptr ≥ ntblsz) then error; return; fi
      dmntbl (1,dmnptr) ← lnth);
      dmntbl (8,dmnptr) ← exptyp;
      dmntbl (9, dmnptr) ← typcl;
      comment: we enter name of variable now
      for i = 1 to lnth do
         dmntbl (i+1, dmnptr) ←token(i);
      od
      for i = 1 to dmnptr do
        if (dmntbl(1,i)=lnth) then
          begin for j = 1 to lnth do
                     if (dmntbl(j+1,i)≠token(j)) then go to out;
                     fi
                od
                entno ← i;
                go to lend;
          end
          out: continue;
        fi
      od
      lend: if (entno = dmnptr) then dmnptr ← dmnptr+1; fi
            entno ← entno * 10+3;
end
```

30. Procedure adjdmn(entno, numarg, arg(1:5), errflg);

```
begin comment: This routine stores arguments of a dimensioned
               variable, if entry no. is given;
      entry ←entno/10;
      if (entry ≥ dmnptr) then error; return; fi
      dmntbl (10, entry) ← numarg;
      for i = 1 to numarg do
          dmntbl (i+10, entry) ← arg(i);
      od
end
```

```
31.       Procedure fchdmn (entno, lnth, token(1:1000),
          exptyp, typcl, numarg, arg(1:5), errflg);
   begin  comment: This routine fetches an entry from dimension
                   var. table, given an entry number;
          if (entno > ntblsz) then error; return; fi
          lnth ← dmntbl (1,entno);
          for i = 1 to lnth do
              token(i) ← dmntbl(i+1, entno);
          od
          exptyp ← dmntbl (8,entno);
          typcl ← dmntbl (9, entno);
          numarg ← dmntbl (10,entno);
          for i = 1 to numarg do
              arg(i) ← dmntbl (i+10, entno);
          od
   end

32.       Procedure enddmn;
   begin  dmntbl(1,dmnptr) ← marker; end

33.       Procedure schdmn (lnth, var(1:10), found, entno);
   begin  comment: Given a name, this routine searches
                   dimensioned variable table, and reports;
          for i = 1 to (dmnptr-1) do
              if (dmntbl (1,i) = lnth) then
                     begin for j = 1 to lnth do
                           if (dmntbl(j+1,i)≠var(j) then go to
                                                          last;
                           fi
                           od
                           entno ← i*10+3; found ← 'true'; return
                     end
                 last: continue;
              fi
           od
           found ← 'false'
   end

34.       Procedure inismp;
   begin  mplptr ← 1; end

35.       Procedure entsmp (lnth, token(1:1000), exptyp,
          typcl, entno, errflg);
   begin  comment: This routine enters a simple variable into
                   table and returns entry number;
```

```
        if (lnth > 6) then error; return; fi
        if (mplptr > mplsze) then error; return; fi
        mpltbl(1, mplptr) ← lnth;
        mpltbl (8, mplptr) ← exptyp;
        mpltbl (9, mplptr) ← typcl;
        for i = 1 to lnth do
            mpltbl(i+1, mplptr) ← token(i);
        od
        for i = 1 to mplptr do
            if (mpltbl(1,i) = lnth) then
                begin for j = 1 to lnth do
                        if (mpltbl(j+1,i)≠token(j)) then
                            begin go to out; end
                        fi
                      od
                      entno ← i;
                      go to last;
                end
                out: continue;
            fi
       od
       last: if(entno=mplptr) then  mplptr ← mplptr+1; fi
             entno ← entno*10+4;
end
```

```
36.    Procedure fchsmp (entno, lnth, token(1:1000),exptyp,
       typcl, errflg);
  begin comment: This routine gets an entry of simple variable
                 table if entry number is given;
       if (entno > mplsze) then error; return fi
       lnth ← mpltbl(1,entno);
       for i = 1 to lnth do
           token(i) ← mpltbl(i+1,mplptr);
       od
       exptyp ← mpltbl(8,entno);
       typcl ← mpltbl (9,entno);
  end
```

```
37.    Procedure endsmp;
  begin mpltbl(1, mplptr) ← marker; end
```

```
38.    Procedure inisfn;
  begin sfnptr ← 1; end
```

```
39.     Procedure entsfn(lnth, token(1:1000),entno, errflg,
        flag);
   begin comment: This routine enters the name of subprogram/
                  function into the table and returns entry
                  number:
                  Flag indicates whether newly entered or
                  previously present;
        flag ← 'true';
        if (sfnptr ≥ mtblsz) then error; return fi
        if (lnth > 6) then error; return fi
        sfntbl (1,sfnptr) ← lnth;
        for i = 1 to lnth do sfntbl(i+1,sfnptr) ← token(i);od
        comment: new we search the table to see if already
                  present;
        for i = 1 to sfnptr do
            if (sfntbl(1,i)=lnth) then
                begin for j = 1 to lnth do
                          if (sfntbl(j+1,i) ≠ token(j)) then
                                               go to out;
                          fi
                      od
                      entno ← i;
                      go to last;
                end
                out: continue;
            fi
        od
        last: if (entno = sfnptr) then
                      begin flag ← 'false';
                            sfnptr ← sfnptr+1; end
              fi
              entno ← entno*10+5;
   end

40.     Procedure adjsfn(entno, numarg, arg(1:20), defbit,
        defent, chkbit, errflg);
   begin comment: This puts arguments etc. of a subprogram/
        function if entry number of it is given;
        entry ← entno/10;
        if (entry ≥ sfnptr) then error; return fi
        if (chkbit≠1) then
            begin sfntbl (8,entry) ← numarg; k ← 1;
                  while (k ≤ numarg) do
                      strptr ← 9; klim ← min(k+8,numarg);
                      while (k ≤ klim) do
                           sfntbl(strptr,entry) ← arg(k);
                           k ← k+1; strptr ← strptr+1;
                      od
                      if (k ≤ numarg) then sfntbl(18,entry) ←
                                                  kntmrk;
                                      else sfntbl(18,entry) ←
                                                  nocnt;
                      fi
```

```
                       entry ← entry+1;
                    od
                    ent ← entno/10;
                    sfntbl(19,ent) ← defbit;
                    if (defbit=1) then sfntbl(20,ent)←defent;
                    fi
                    sfnptr ← entry;
              end
              else if (sfntbl (8,entry)≠numarg ∨ sfntbl(20,entry)
                                                  ≠defent)
                                  then error;
                   fi
         fi
   end

41.      Procedure fchsfn(entno,lnth,token(1:1000),numarg,
         arg(1:20), defbit, defent, errflg);
   begin comment: This routine fetches an entry from the
                  table if entry number is given;
         if (entno > mtblsz) then error; return fi
         lnth ← sfntbl (1,entno);
         for i=1 to lnth do token(i) ← sfntbl(i+1,entno); od
         numarg ← sfntbl(8, entno);
         defbit ← sfntbl(19, entno);
         if (defbit=1) then defent←sintbl(20,entno); fi
         nptr ← entno; k ← 1;
         while (k ≤ numarg) do
            strptr ← 9; klim ← min(k+8,numarg);
            while (k ≤ klim) do
               arg(k) ← sfntbl(strptr,nptr);
               k ← k+1; strptr ← strptr+1;
            od
            if (k ≤ numarg) then
                    if(sfntbl(18,nptr)≠kntmrk)then error; return
                    fi
               else if (sfntbl(18,nptr)≠nocnt)then error;return
                    fi
            fi
            nptr ← nptr+1;
         od
   end

42.     Procedure endsfn;
   begin sfntbl(1,sfnptr) ← marker; end
```

```
43.        Procedure schsfn(lnth, var(1:10),found,entno);
  begin  comment: Given name, this routine searches subprog./
                  function table, and reports back;
         for i=1 to (sfnptr-1) do
            if(sfntbl(1,i)=lnth) then
                  begin for j=1 to lnth do
                             if (sfntbl(j+1,i)≠var(j) then
                                                 go to last;
                             fi
                        od
                        entno ← i*10+5; found ← 'true';return
                  end
                  last: continue;
            fi
         od
         found ←'false';
  end

44.        Procedure inistn;
  begin istmpt ← 1; end

45.        Procedure entstn (nmbr, entno,errflg);
  begin comment: This routine enters a statement number into
                 the table and returns entry number;
        if (istmpt ≥ ktblsz) then error; return fi
        if (nmbr≠ -9999) then
              begin comment: We search the number to see if
                                 already present;
                    for i = 1 to (istmpt-1) do
                        if (istmtb(1,i)=nmbr)then entno ←
                                                  i*10+6;
                                                  return fi
                    od
              end
        fi
        istmtb (1,istmpt) ←nmbr;
        entno ← istmpt*10+6;
        istmpt←istmpt+1;
  end

46.        Procedure adjstn (entno,fmtflg,num,errflg);
  begin comment: Given an entry number, this routine adjusts
                 links;
        nentry ← entno/10;
        if (nentry ≥ istmpt) then error; return fi
        istmtb (2,nentry) ←fmtflg;
        istmtb (3,nentry) ← num;
  end
```

```
47.       Procedure fchstn (entno, stno, fmtflg, link, errflg);
  begin comment: This routine fetches an entry, when entry
                 number is given;
        if (entno ≥ ktblsz) then error; return fi
        stno ← istmtb (1,entno);
        fmtflg ← istmtb (2,entno);
        link ← istmtb (3,entno);
  end

48.       Procedure endstn;
  begin istmtb (1,istmpt) ← marker; end

49.       Procedure maxflb (iflb);
  begin comment: This routine gives flow blocks used upto
                 new;
        ichek ← istmpt-1;
        iflb ← 0;
        if (ichek≠0) then
            begin for i = 1 to ichek do
                      if (istmtb(2,i)≠1) then
                          begin if (istmtb(3,i)>iflb)then
                                         iflb←istmtb(3,i);
                                fi
                          end
                      fi
                  od
            end
        fi
  end
50.       Procedure stnosz (itblsz);
  begin comment: This gives size of stmt no. table;
        itblsz ← istmpt-1;
  end

51.       Procedure iniseg (nflbno);
  begin comment: This initialises segment header;
        nsghdr(1) ← main; nsghdr(2) ← lnkcmt;
        nsghdr(3) ← nflbno;
  end
```

```
52.          Procedure adjseg(class,length,token(1:1000),sbrfnc,
             exptyp, typcl, errflg);
    begin comment: This adjusts segment header, when it is
                   found that segment header is subroutine
                   or function;
         sveflb <-- nsghdr(3);  svecmt <-- nsghdr(2);
         nsghdr(1) <-- sbrfnc;  nsghdr(2) <-- exptyp;
         if (exptyp=1) then nsghdr(3)<-typcl; fi
         fchtkn (class, length, token(1:1000));
         if (class ≠ idcls) then error; return fi
         nsghdr(4) <-- length;  locptr <--length+5;
         for i=1 to length do nsghdr(i+4)<-- token(i); od
         nargpt <-- locptr;
         locptr <-- locptr+1;
         numarg <-- 0;
         fchtkn (class, length, token (1:1000));
         if (class ≠ keol) then
              begin if (class ≠ lparc l) then error; return fi
                    fchtkn(class,length,token(1:1000));
                    if (class≠idcls ∧class≠intcls)then error;
                                                       return
                    fi
                    for i=1 to length do
                         nsghdr (locptr) <-- token(i);
                         locptr <-- locptr+1;
                    od
                    numarg <-- numarg+1;
                    fchtkn (class, length, token(1:1000));
                    while (class ≠ rparcl) do
                       if (class ≠ kmmacl) then error; return
                       fi
                       nsghdr (locptr) <-- token(1);
                       locptr <-- locptr+1;
                       fchtkn (class, length, token(1:1000));
                       if (class≠idcls ∧class≠intcls) then
                                           error; return fi
                       for i=1 to length do
                           nsghdr(locptr)<-- token(i);
                           locptr <-- locptr+1;
                       od
                       numarg <-- numarg+1;
                       fchtkn (class,length,token(1:1000));
                    od
                    fchtken (class,length,token(1:1000));
                    if (class≠keol) then error; return fi
              end
         fi
         nsghdr (locptr)<-- svecmt;
         nsghdr (locptr+1)<--sveflb;
         nsghdr (locptr+2)<- marker;
         nsghdr (nargpt)<- numarg;
  end
```

```
53.        Procedure fchseg(nseg(1:50));
   begin comment: This routine gets contents of segment header;
         i ← 0;
         Repeat
             i ← i+1;
             nseg(i) ← nsghdr(i);
         until (nseg(i) = mrkr)
   end

54.      Procedure iniflb (iflbno);
   begin iflbno ← 0; end

55.      Procedure getflb (iflbno, errflg);
   begin comment: This routine gives out a new flow block number;
         if (iflbno ≥ noflb) then error; return fi
         iflbno ←iflbno+1;
         nflblk (1,iflbno) ← 2;
   end

56.      Procedure entflb (type,stno,kdbno,iflbno);
   begin comment: This routine enters type of segment etc. for
                  which flow block is being used;
         nflblk(2,iflbno) ← type;
         nflblk(3,iflbno) ← stno;
         nflblk(4,iflbno) ← kdbno;
         nflblk(1,iflbno) ← 5;
   end

57.      Procedure nxtflb (iflbno, nentry, entry);
   begin comment: This routine enters successor of this flow
                  block;
         if (nflblk(1,iflbno) ≥ iflbsz) then
                begin nflblk (iflbsz, iflbno) ← kntmrk;
                      getflb (iflbno, errflg);
                end
         fi
         itm ← nflblk (1,iflbno);
         nflblk (itm, iflbno) ← nentry;
         nflblk (1,iflbno) ← nflblk(1,iflbno)+1;
  end

58.      Procedure endflb (iflbno);
  begin  itm ← nflblk(1,iflbno);
               nflblk(itm,iflbno) ← marker;
               nflblk(1,iflbno) ← nflblk(1,iflbno)+1;
  end
```

```
59.        Procedure chkflb (iflbno, wound);
   begin comment: This routine checks whether flowback is
                  wound or not;
         wound ← 'flase';
         itm ← nflblk (1,iflbno);
         if (nflblk (itm-1, iflbno) = makker) then wound ← 'true'
         fi
  end

60.        Procedure gtkdno (nocdbl, iflb);
  begin comment: This routine gets code-block of a flowblock
                 given;
        nocdbl ← 0;
        if (iflb≠0) then nocdbl ← nflblk(4,iflb); fi
  end

61.     Procedure inicdb (:kdbno);
  begin kdbno ← 0; end

62.     Procedure getcdb (kdbno);
  begin kdbno ← kdbno+1; kdbptr ← 1; end

63.     Procedure entcdb (kdbno, class, length, token(1:1000),
        entno);
  begin comment: This routine enters a token into code block;
        l ← length;
        if (l=0) then l←l; fi
        spavlb ← kdbsz-kdbptr;
        if (spavlb < l+2) then
                 begin kdblk (kdbptr,kdbno) ← knt;
                       getcdb (kdbno);
                 end
        fi
              kdblk(kdbptr,kdbno) ← class;
        kdblk (kdbptr+1, kdbno) ← length;
        kdbptr ← kdbptr+2;
        if (length=0) then
               begin kdblk (kdbptr,kdbno) ← entno;
                     kdbptr ← kdbptr+1;
               end
          else for i = 1 to length do
                   kdblk (kdbptr, kdbno) ← token(i);
                   kdbptr ← kdbptr+1;
               od
        fi
  end
```

```
64.        Procedure cheokd (kdend, kdbno);
   begin comment: This checks to see a codeblock end has
                  come;
         kdend ← 'false';
         if (kdblk (kdbptr, kdbno)=mrkr) then kdend ←'true' fi
   end

65.      Procedure endcdb (kdbno);
   begin kdblk (kdbptr, kdbno) ← marker;
         kdbptr ← kdbptr+1;
   end

66.      Procedure kdtkn (kdbno, class, length, token(1:1000));
   begin comment: This routine gets next token of code block;
         class ← kdblk (kdbptr, kdbno);
         if (class≠knt) then
              begin length ← kdblk (kdbptr+1, kdbno);
                    kdbptr ← kdbptr+2;
                    l ← length;
                    if (l=0) then l←l; fi
                    for i=l to l do
                        token(i) ← kdblk(kdbptr, kdbno);
                        kdbptr ← kdbptr+1;
                    od
              end
        fi
   end

67.      Procedure inidcl;
   begin idclpt ← 1; end

68.      Procedure entdcl (class,length,token(1:1000),entno,
         errflg);
   begin comment: This routine enters  a token in declarative
         block;
         ispavl ← idclsz - idclpt;
         l ← length;
         if (l=0) then l=l; fi
         if (ispavl < l+2) then error; return fi
         idclbl (idclpt) ← class;
         idclbl (idclpt+1) ← length;
         idclpt ← idclpt+2;
         if (length=0) then
               begin idclbl(idclpt) ← entno;
                     iddlpt ← idclpt+1;
               end
            else for i = 1 to length do
                    idclbl (idclpt) ← token(i);
                    idclpt ← idclpt+1;
                 od
         fi
  end
```

```
69.       Procedure enddcl;
   begin idclbl(idclpt)<--marker; end

70.       Procedure chendl (dclend);
   begin comment: This routine checks to see if end of
                  tokens achieved;
         dclend <-- 'false'
         if (idclbl(idclpt) = mrkr) then dclend <-- 'true' fi
   end

71.       Procedure dcltkn (class, length, token (1:1000));
   begin comment: This routine gets next token from dec-
                  larative block;
         class <--idclbl (idclpt);
         length<--idclbl (idclpt+1);
         idclpt<--idclpt+2;
         l <-- length;
         if (l=0) then l<-- 1; fi
         for i=1 to l do
             token(i)<--idclbl (idclpt);
             idclpt <-- idclpt+1;
         od
   end

72.       Procedure prsgmt (temp(1:72), iflbno, kdbno, errflg,
          endfle, deflg);
   begin comment: This routine processes a segment;
         initialise tables, blocks, segment, stack etc;
         get flow and code blocks;
         newlxi, fchtkn(class,length, token(1:1000));
         type<-- 0; sbrtfn<--iabs (class)-10+1; exptyp<--0;
         if (class = sbrtcl ∨ class = fncls ∨ class = ibldcl)
            then begin adjseg (class, length, token (1:1000),
                                  sbrtfn, exptyp, typcl, errflg);
                        type<--sbrtfn;
                        dolxi (temp(1:72), endfle); newlxi;
                        fchtkn (class, length, token (1;1000));
                 end
            else kls <-- iabs (class);
                 comment: check for function subprogram like
                          'integer function' etc;
                 if (kls ≥ 120 ∧ kls ≤ 123) then
                        begin exptyp <-- 1; typcl<--kls/100;
                              sbrtfn <-- 2;
                              fchtkn (class, length,
                              token (1:1000));
                              adjseg (class, length,token
                                     (1:1000), sbrtfn,
                                     exptyp, typcl,errflg);
                              dolxi (temp(1:72); endfle);
                                    newlxi;
```

```
                                    fchtkn (class, length, token
                                                (1:1000));
                              end
                       fi
       fi
       entstn (-9999, entno, errflg);
       entflb (type, -9999, kdbno, iflbno);
       adjsln (entno, 0, iflbno, errflg);
       entcdb (kdbno, kntncl, 8, kntnue, ndum);
       entcdb (kdbno, keol, 1, mrkr (1:1), ndum);
       while (class ≠ endcls) do
          prstmt (class, length, token (1:1000), iflbno,
                  kdbno, type, idumno, errflg);
          dolxi (temp (1:72), endfle); newlxi;
          fchtkn (class, length, token (1:1000));
       od
       comment: we got an END stmt.;
       chkflb (iflbno, wound);
       if (¬  wound) then endflb (iflbno) fi
       chkstk (empty);
       if (¬ empty) then error; fi
   end

73.        Procedure prstmt (class, length, token(1:1000),
           iflbno, kdbno, type, idumno, errflg);
   begin   comment: This processes a statement of the segment;
           stno ← 0;
           if (class = cmtcls) then
                  begin entlne (class, length, token (1:1000),
                                1, entno, errflg);
                        entcdb (kdbno, class, 0, token(1:1000),
                                entno);
                        entcdb (kdbno, keol, 1, mrkr(1:1),
                                ndum);
                  end
             else if (class ε decicls) then prdecl (class,
                                            length, token
                                            (1:1000), errflg);
                       else comment: check for stmt. function;
                            if (class = idcls) then
                                begin for i = 1 to length do
                                          jvar(i) ← token(i);
                                      od
                                      jlnth ← length;
                                      fchtkn (class, length,
                                             token(1:1000));
                                      if (class≠lparcl) then
```

```
                        begin if (class≠equlcl)
                                 then error;
                                      return
                              fi
                              comment: we pro-
                                       cess
                                       assign-
                                       ment
                                       stmt.;
                              enter rest of
                               stmt. into code
                               block, after
                               entering into
                               tables;
                        end
                   else schdmn(jlnth, jvar
                               (1:10),found,
                                entno);
                        if (found) then
                             begin enter rest
                                   of stmt.in
                                   to code
                                   block after
                                   entering
                                   variable/
                                   const. into
                                   tables;
                             end
                             else comment:
                                  we got stmt.
                                  function;
                                  enter name
                                  of function
                                  and its argu-
                                  ments in
                                  subprogram/
                                  function name
                                  table; enter
                                  stmt. into
                                  code block;
                       fi
                fi
           end
       else comment: we check forstmt. no.
           if (class=stnocl) then
                 begin stno ← token(1);
                       entstn (stno,entno,
                               errflg);
                       nent ← entno;
                       fchtkn (class, length,
                       token (1:1000));
                       if (class=fmtcls) then
```

```
                                                  entlne(class,
                                                      length,
                                                      token(1:1000),
                                                      7,entno,
                                                      errflg);
                                                  adjstn(nent, 1,
                                                      entno,errflg);
                                                  return
                                              fi
                                           end
                                     fi
                                     prblst (stno,nent,class,
                                              length,token(1:1000),
                                              iflbno,kdbno,type,
                                              idumno,errflg);
                              fi
                        fi
              fi
     end

74.          Procedure prblst(stno,nent,class,length,token
             (1:1000), iflbno, kdbno,type,idumno,errflg);
     begin comment: This processes a blockable statement;
           if (stno≠0) then
                 begin chkflb (iflbno,wound);
                       if (¬ wound) then
                              begin nxtflb (iflbno, nent,errflg);
                                    endcdb (kdbno); endflb(iflbno).
                              end
                       fi
                       getflb (iflbno, errflg); getcdb (kdbno);
                       entflb(type,stno,kdbno, iflbno);
                       adjstn (nent, 0, iflbno, errflg);
                       entcdb(kdbno, istncl,0,token(1:1000),
                               nent);
                 end
           fi
           comment: now we process rest of statement excluding
                    stmt. no. ;
           case class of
              declass: procdo(class,length,token(1:1000),iflbno,
                              kdbno, type, idumno, errflg);
                noncontrol  }
                   and      }
                logical IF  } : case class of
                class       }     non-        if (class≠kallcl)
                                  control:      thenenter the
                                                statement into
                                                code block
                                                after entering
                                                variables/
                                                constants into
                                                respective
                                                tables;
```

```
                                    else enter
                                    stmt.into
                                    code block;
                                    enter the
                                    subroutine
                                    called in
                                    subprogram/
                                    function table
                                    alongwith
                                    arguments;
                              fi
                  log IF cls: entcdb(kdbno,
                                    class,length,
                                    token(1:1000)),
                                  if (class≠lparcl)
                                    then error;
                                    return fi
                                  endtcdb (kdbno,
                                  class,length,
                                  token(1:1000),
                                  ndum);
                                  fchtkn (class,
                                  length,token
                                  (1:1000));
                                  entxpr(class,
                                   length, token
                                   (1:1000),kdbno,
                                      errflg);
                                   nsvebl←iflbno;
                                   entcdb(kdbno,
                                   keol,1,mrkr(1:1),
                                   ndum);
                                   endcdb(kdbno);
                                   entstn(-9999,
                                   entno, errflg);
                                   nxtflb(iflbno,
                                   entno, errflg);
                                   getflb(iflbno,
                                   errflg);
                                   getcdb(kdbno);
                                   entflb(type,-9999,
                                   kdbno,iflbno);
                                   adjstn(entno,0,
                                   iflbno,errflg);
                                   prtgt(class,
                                   length,token
                                   (1:1000),nsvebl,
                                    kdbno,iflbno,
                                    type,errflg);
               end
```

```
                    comment: now we check whether stmt.
                             ends range of a DO;
                    chendo(stno,dument,parnt,fnent,
                           stpent,doend,errflg);
                    while (  ¬ doend) do
                       enter parnt = parnt+stpent and
                             IF(parnt ≤ fnent) go to
                                               dument
                       into the code block;
                       nxtflb (iflbno, dument,errflg);
                       entstn (-9999,entno,errflg);
                       nxtflb (iflbno,entno,errflg);
                       endcdb (kdbno); endflb(iflbno);
                       getflb (iflbno,errflg); getcdb
                                                 (kdbno);
                       adjstn (entno,0,iflbno,errflg);
                       entflb (type, -9999, kdbno,errflg);
                       entcdb (kdbno,kntncl,8,kntnue
                               (1:8), ndum);
                       entcdb (kdbno, keol,1,mrkr(1:1),
                                ndum);
                       chendo (stno, dument,parnt,fnent,
                               stpent,doend,errflg);
                    od
          Control)
          other  )
          than   )  : prctl (class,length,token(1:1000),
          log IF,)             iflbno,kdbno,errflg);
          DO,END )

      end
 end

75.    Procedure prcdo (class,length,token(1:1000);iflbno,
       kdbno, type, idumno, errflg);
  begin comment: This routine processes a DO statement;
       get all parameters of DO stmt.;
       comment: do range, parnt,inent,fnent,stpent contain
                all the values.  If step not given then it
                is taken as one;
       enter 'parnt=inent' in code block;
       getdum(idumno);
       entstn(idumno, entno,errflg);
       nxtflb(iflbno,entno,errflg);
       endflb(iflbno); endcdb (kdbno);
       getflb (iflbno, errflg); getcdb(kdbno);
       entflb (type, idumno, kdbno, iflbno);
       adjstn (entno, 0, iflbno, errflg);
       pshstk (dornge,entno,parnt,fnent,stpent,errflg);
       entcdb(kdbno,intcls,0,token(1:1000),entno);
       entcdb(kdbno,kntncl,8,kntnue(1:8),ndum);
       entcdb(kdbno,keol,1,mrkr(1:1),ndum);
  end
```

```
76.         Procedure prctl(class,length,token(1:1000),iflbno,
            kdbno,errflg);
   begin    This routine processes control stmts. other than
            log. IF, DO and END;
          case class of
             arith. IF}  entcdb (kdbno,class,length,token
               class  }              (1:1000), ndum);
                         fchtkn(class,length,token(1:1000));
                         if (class≠lparcl) then error; return fi
                         entcdb (kdbno,class,length, token
                                     (1:1000),ndum);
                         fchtkn (class, length, token(1:1000));
                         entxpr (class, length, token(1:1000),
                                      kdbno,errflg);
                         if (class≠intcls) then error; return fi
                         entstn (token(1), entno, errflg);
                         entcdb (kdbno,class,0,token(1:1000),
                                      entno);
                         nxtflb (iflbno,entno,errflg);
                      for i = 1 to 2 do
                            fchtkn (class,length,token(1:1000));
                            if (class≠kmmacl) then error; return
                            fi
                            entcdb(kdbno,class,length,token
                                     (1:1000),ndum);
                            fchtkn (class,length,token(1:1000));
                            if (class≠intcls) then error;return
                            fi
                            entstn (token(1),entno,errflg);
                            entcdb(kdbno,class,0,token(1:1000),
                                     entno);
                            nxtflb (iflbno, entno, errflg);
                      od
                      fchtkn(class,length,token(1:1000));
             go to    entcdb (kdbno,class,length,token(1:1000),
             class:           ndum);
                      case class of
                       assgnd}
                       go to }  entsmp (length,token(1:1000),
                       cl.   }  0,ndum,entno,errflg);
                                entcdb (kdbno,class,0,token
                                        (1:1000),entno);
                                fchtkn(class,length,token
                                        (1:1000));
                                if (class≠kmmacl) then error;
                                                        return
                                fi
                                entcdb (kdbno, class,length,
                                        token(1:1000), ndum);
                                fchtkn(class,length,token
                                       (1:1000));
```

```
            if (class≠lparcl) then error;
                                   return
            fi
            entcdb (kdbno,class,length,
                   token(1:1000),ndum);
            fchtkn(class,length,token(1:1000))
            entnmb (class,length, token
                    (1:1000), kdbno,iflbno,
                     errflg);
ordnry )
go to  }: entstn(token(1),entno,errflg);
cl     )  entcdb (kdbno,class,0, token
                   (1:1000),entno);
          nxtflb (iflbno,entno,errflg);
          fchtkn(class,length,token
                   (1:1000));
computed )
go to    }entcdb(kdbno,class,length,
cl.      )token(1:1000),ndum);
          fchtkn(class,length,token
                  (1:1000));
          entnmb(class,length,token
                  (1:1000), kdbno,
                   iflbno,errflg);
          if (class≠kmmacl) then error;
                                 return
          fi
          entcdb(kdbno,class, length,
                  token(1:1000),ndum);
          fchtkn (class,length,token
                    (1:1000));
          if (class ≠ idcls) then error;
                                  return
          fi
          entsmp (length,token(1:1000),
                   0,ndum,entno,errflg);
          entcdb (kdbno,class,0,token
                   (1:1000), entno);
          fchtkn (class,length,token
                   (1:1000));

end
```

```
                        stop/ }
                       return } :  entcdb(kdbno,class,length,
                       class  }    token (1:1000),ndum);
                                   nxtflb (iflbno,0,errflg);
                                   fchtkn (class,length,token(1:1000));
                                   if (class=intcls) then
                                         begin entcns (class,length
                                                        token(1:1000),
                                                        entno,errflg);
                                               entcdb(kdbno,class,0,
                                                       token(1:1000),
                                                       entno);
                                               fchtkn (class,length,
                                                       token(1:1000));
                                         end
                                   fi
                end
                if (class≠keol) then error; return fi
                entcdb (kdbno,class,length,token(1:1000),ndum);
                endflb(iflbno); endcdb (kdbno);
            end

77.         Procedure prtgt (class, length, token(1:1000),
            nsvebl, kdbno,iflbno,type,errflg);
      begin This processes target of logical IF;
            if (class ∉ non-control ∧ class ∉ control o/t log.
                IF, DO, END) then error; return fi
            case class of
              non-control: enter the stmt. into code block;
                           entstn (-9999, entno,errflg);
                           nxtflb (iflbno,entno,errflg);
                           endcdb (kdbno);nxtflb(nsvebl,entno,errflg);
                           endflb (iflbno); endflb (nnsvebl);
              control:     prctl (class,length,token(1:1000),
                                  iflbno,kdbno,errflg);
                           entstn (-9999, entno, errflg);
                           nxtflb (nsvebl,entno,errflg);
                           endflb (nsvebl);
            end
            getflb (iflbno, errflg); getcdb(kdbno);
            entflb (type, -9999, kdbno, iflbno);
            adjstn (entno, 0, iflbno, errflg);
            entcdb (kdbno, kntncl, 8, kntnue (1:8), ndum);
            entcdb (kdbno, keol,1,mrkr(1:1),ndum);
      end
```

```
78.     Procedure prdecl (class, length, token(1:1000),
                          errflg);
  begin comment: This routine processes all declarative
                 statements;
        if (class=ixtncl) then
              begin enter stmt. into declarative block,
                    after entering variables in subprogram/
                    function table;
              end
           else if (class=cmncl v class=eqc l) then
                     begin enter stmt. as it is into
                           declarative block, but enter
                           dimensioned var. of COMMON
                           stmt. into corresponding table;
                     end
                  else if (class≠datacl) then
                            begin enter stmt. into dec-
                                  larative block, after
                                  entering variables into
                                  simple/dimensioned var.
                                  table;
                            end
                         else enter data stmt. into declara-
                              tive block, after entering
                              variables/constants into
                              their respective tables;
                  fi
           fi
        fi
  end

79.      Procedure entvar (class,length,token(1:1000),dclbit,
                           ntypcl, errflg);
  begin  Comment: This routine enters a variable of decl. stmt.;

         for i = 1 to length do nsvar(i)←—token(i); od
         nsvlnt ←— length;
         fchtkn(class, length, token(1:1000));
         if (class = lparcl) then
              begin entdmn (nsvlnt,nsvar (1:10),dclbit,
                            ntypcl,entno,errflg);
                    entdcl (idcls, 0, token(1:1000),entno,
                                errflg);
                    fchtkn(class, length, token(1:1000));
                    getarg (class, length, token(1:1000),
                              numarg, arg(1:5), 0, errflg);
                    adjdmn (entno,numarg,arg(1:5), errflg);
                    fchtkn(class, length, token(1:1000));
              end
           else entsmp (nsvlnt,nsvar (1:10),dclbit,ntypcl,
                            entno,errflg);
                entdcl (idcls, 0, token(1:1000), entno,
                             errflg);
         fi
  end
```

```
80.       Procedure entid (class,length,token(1:1000),entno,
                          kdbno,ndclbt,errflg);
   begin for i=1 to length do nsvar(i)← token(i); od
         nsvlnt ← length;
         fchtkn (class, length, token (1:1000));
         if (class≠lparcl) then entsmp (nsvlnt, nsvar(1:10);
              0, ndum,entno,errflg);
             else schdmn (nsvlnt,nsvar(1:10),found,entno);
                  if (¬ found) then
                      begin schsfn (nsvlnt,nsvar(1:10),found,
                                 entno);
                            if (¬ found) then
                                 begin prcfun(nsvlnt,nsvar(1:10),
                                        class,length,
                                        token(1:1000),
                                        entno,kdbno,
                                        errflg);
                                   return
                                 end
                            fi
                      end
                  fi
         fi
         if (ndclbt=1) then entdcl(idcls,0,token(1:1000),entno,
                                 errflg);
                      else entcdb(kdbno,idcls,0,token(1:1000),
                                 entno);
         fi
   end

81.       Procedure prcfun(nsvlnt, nsvar(1:10),class,length,
                          token(1:1000), entno,kdbno,errflg);
  begin   comment: This ~~xxx~~ routine processes a function call;
          entsfn(nsvlnt, nsvar(1:10),entno,errflg,flag),
          nsvent ← entno;
          entcdb (kdbno,idcls, 0, token(1:1000), entno);
          entcdb (kdbno, class,length, token(1:1000),ndum);
          get arguments of function call, after entering them
          in tables;
          adjsfn (nsvent, noarg, arg(1:20), 2,ndum,0,errflg);
          fchtkn (class, length, token (1:1000));
  end

82.       Procedure inidum (no);
  begin   no ← 0; end

83.       Procedure getdum (no);
  begin   no ← no-1; end
```

```
84.        Procedure entxpr (class,length,token(1:1000),
           kdbno, errflg);
   begin parnct ← 1;
         if (class=lparcl) then parnct ← parnct+1; fi
         while (parnct≠0) do
            if (class=idcls) then entid (class,length,
               token (1:1000), entno,kdbno, 0,errflg);
              else if (class=int.cls ∨ class=realcl) then
                     begin entcns (class,length,token
                                   (1:1000),entno,
                                    errflg);
                           entcdb (kdbno,class,0,token
                                   (1:1000),entno);
                           fchtkn(class,length,token
                                   (1:1000));
                     end
                   else entcdb (kdbno,class,length,
                                token(1:1000),ndum);
                        fchtkn (class,length,token(1:1000));
                  fi
            fi
            if (class=lparcl) then parnct ←parnct+1; fi
            if (class=rparcl) then parnct ← parnct-1; fi
         od
         entcdb (kdbno,class,length,token(1:1000), ndum);
         fchtkn (class,length,token(1:1000));
   end

85.        Procedure dolxi (temp(1:72),endfle);
   begin clstmt (stmt(1:700), temp(1:72), endfle,lnstmt,error);
         i ← 0; klch ← 0; klass2 ← 0;
         while (klch≠ klblnk) do nxtchr (stmt (1:700),i); od
         lexcal (stmt (1:700),i,lnstmt, error);
   end

86.        Procedure getarg (class,length,token(1:1000),numarg,
           arg, jdclbl, errflg);
   begin numarg ← 0; state ← 1;
         loop: kls ← 5;
         if (class=idcls ∨ class=intcls) then kls ← class+1;
         fi
         if (class=rparcl ∨class=kmmacl) then kls ←class-5;fi
         ntemp ← fsmch (stak, kls);
         nact ← ntemp/10;
         nstate ← ntemp-nact*10;
         case nact of
              id or
              censt: numarg ←numarg+1;
                     if (class=idcls) then entsmp (length,
                                             token(1:1000),0,
                                             ndum,ento,errflg);
                                      else entcns (class,
                                                    length,
                                                   token(1:1000);
                                                   entno, errflg);
                     fi
```

```
                arg (numarg) ← entno;
                if (jdclbl=1) then entdcl (class, length,
                                           token(1:1000),
                                           ndum,errflg); fi
                fchtkn (class,length,token(1:1000));
                state ← nstate;
                go to loop;
          comma:if (jdclbl=1) then entdcl (class,length,token
                                           (1:1000), ndum,
                                            errflg); fi
                fchtkn (class,length,token(1:1000));
                state ← nstate;
                go to loop;
           rpar: return
         others: error;
      end
   end
```

87. Procedure entnmb (class,length,token(1:1000),kdbno, iflbno,errflg);

```
  begin state ← 1;
        loop:kls ← 4;
             if (class=intcls) then kls ← class; fi
             if (class=rparcl ∨ class=kmmacl) then kls ← class-6;
             fi
             ntem ← mchtb (state, kls);
             nxt ← ntem/10;
             nstate ← ntem-nxt*10;
             case nxt of
                  comma:entcdb(kdbno,class,length,token(1:1000)
                               ndum);
                        fchtkn (class,length,token (1:1000));
                        state ← nstate;
                        go to loop;
                  stmt. no.: entstn (token(1),entno,errflg);
                             entcdb(kdbno,class,0,token(1:1000),
                                    entno);
                             nxtflb(iflbno,entno,errflg);
                             fchtkn (class,length,token(1:1000));
                             state ← nstate;
                             go to loop;
                   rpar:entcdb (kdbno,class,length,token(1:1000),
                                ndum);
                        fchtkn(class,length,token (1:1000));
                 others:error;
             end
   end
```

88. Procedure inistk;

```
   begin istkpt ← 0; end
```

```
89.         Procedure pshstk (dornge,entno,parnt,fnent,
                              stpent,errflg);
   begin   if (istkpt ≥ istksz) then error; return fi
           istkpt ← istkpt+1;
           idostn (istkpt) ← dornge; jent(istkpt) ← entno;
           ipar (istkpt) ← parnt; ifin (istkpt) ← fnent;
           istep (istkpt) ← stpent;
   end

90.         Procedure chkstk (empty);
   begin   empty ← 'flase';
           if (istkpt=0) then empty ← 'true'; fi
   end

91.         Procedure chendo (stno,dument,parnt,fnent,
                              stpent,doend,errflg);
   begin   doend ← 'false';
           if (istkpt > 0) then
               begin if (stno=idostn(istkpt) then
                          begin dument ← jent (istkpt);
                                parnt ← ipar(istkpt);
                                fnent ← ifin (istkpt);
                                stpent ← istep (istkpt);
                                doend ← 'true';
                                istkpt ← istkpt-1;
                          end
                     fi
               end
           fi
   end

92.         Procedure inibuf (j);
   begin   j ← 0; end

93.         Procedure entbuf (khar,j);
   begin   if (j=72) then prtbuf;
                          for i=1 to 5 do motbf(i) ← blank; od
                              j ← 6; motbf(j) ← kntmrk;
           fi
           j ← j+1;
           motbf(j) ← khar;
   end

94.         Procedure prtbuf;
   begin   if buffer does not contain 'CONTINUE' stmt. then
                                             print buffer; fi
   end

95.         Procedure bufcmf (entno,j);
   begin   fchlne(class,length,token(1:1000),entno,errflg);
           for i = 1 to length do entbuf (token(i),j); od
   end
```

```
96.        Procedure bufcns (entno,j);
   begin   fchcns (class,length,token(1:1000),entno,errflg);
           if (class ≠ intcls) then for i=1 to length do
                                    entbuf(token(i),j); od
                else if (length≠1) then error; return fi
                     knst ← token(1); sgn ← 0;
                     if (knst < 0) then sgn ← 1; fi
                     knst ← iabs(knst);
                     sepcns(knst,ln,dig(1:20));
                     if (sgn=1) then entbuf(minus,j); fi
                     for i=1 to ln do entbuf (dig(i),j); od
           fi
  end

97.        Procedure bufdmn(entno,j);
  begin    fchdmn (entno,lnth,token(1:1000),exptyp,typcl,
                   numarg,arg(1:5), errflg);
           for i=1 to lnth do entbuf (token(i),j); od
  end

98.        Procedure bufsmp (entno,j);
  begin    fchsmp (entno, lnth,token(1:1000),exptyp,typcl,errflg);
           for i=1 to lnth do entbuf (token(i),j); od
  end

99.        Procedure bufsfn (entno,j);
  begin    fchsfn (entno,lnth,token(1:1000),numarg,arg(1:20),
                    defbit, defent, errflg);
           for i=1 to lnth do entbuf (token(i),j); od
  end

100.       Procedure bufstn (entno,j);
  begin    fchstn (entno,stno,fmtflg,link,errflg);
           sgn ← 0;
           if (stno < 0) then sgn ← 1; fi
           stno ← iabs(stno);
           sepcns (stno, ln,dig(1:20));
           if (sgn=1) then entbuf (minus,j); fi
           for i=1 to ln do entbuf (dig(i),j); od
  end

101.       Procedure sepcns (nmbr, ln,dig(1:20));
  begin    if (nmbr ≤ 9) then ln ← 1; dig(1) ← nmbr*2 ↑ 30+bbbbb;
               else ln ← 0;
                   while (nmbr≠0) do
                      ln ← ln+1;
                      temp(ln) ← nmbr-nmbr/10*10;
                      nmbr ← nmbr/10;
                   od
                   for i = 1 to ln do
                         dig(i) ← temp(ln-i+1)*2 ↑ 30+bbbbb;
                   od
           fi
  end
```

```
102.     Procedure prtseg (errflg);
   begin fchseg (nseg (1:50));
         loc ← 1;
         if (nseg(loc)=main) then return fi
             else j ← 0;
                  for i=1 to 6 do entbuf(blank,j); od
                  if (nseg(loc)=bldata) then enter 'BLOCK
                                              DATA' into
                                              buffer; prtbuf;
                      else if (nseg(loc)=isbrtn) then enter
                                        'SUBROUTINE' in
                                        buffer;
                          else if (nseg(loc)≠functn)
                                   then error; return fi
                               if (nseg(loc+1)=1) then
                                      begin enter INTEGER,
                                            into buffer;
                                      end
                                      enter 'FUNCTION'
                                            in buffer;
                          fi
                          loc ← 4; lnth←nseg(loc); loc ←
                                                   loc+1;
                          for i=1 to 2 do entbuf(blank,j);od
                          for i=1 to lnth do
                              entbuf(nseg(loc),j);
                              loc ← loc+1;
                          od
                          if (nseg(loc)≠0) then
                                 begin enter arguments of
                                       subprogram into buffer;
                                 end
                          fi
                          prtbuf;
                 fi
        fi
  end

103.     Procedure prtdcl(errflg);
   begin inidcl;
        chendl(dclend);
        while (¬ dclend) do
           dcltkn(class,length,token(1:1000));
           if (length=0) then error; return fi
           j ← 0;
           for i=1 to 6 do entbuf (blank,j); od
           for i=1 to lnth do entbuf (token(i),j); od
           for i=1 to 2 do entbuf (blank,j); od
           dcltkn (class, length,token(1:1000));
           while (class≠kecl) do
```

```
            if (class≠intcls ∧ length≠0) then for i=1
                                                to length do
                                                entbuf(token
                                                (i),j); od
                 else if (class=intcls) then
                         begin nmbr ← token(1);
                               sepcns (nmbr, ln,dig(1:
                                                   20));
                               for i=1 to ln do entbuf
                                     (dig(i),j); od
                         end
                      else entno ← token(1)/10;
                           nxt←token(1) - entno*10;
                           enter simple or dimensioned
                               var in buffer;
                           comment: If dimensioned var.
                                    then arguments are
                                    also stored;
                   fi
            fi
            dcltkn(class,length,token(1:1000));
          od
          enter blanks in buffer at necessary places;
          prtbuf;
          chendl (dclend);
        od
  end

104.    Procedure prntcd (errflg);
  begin comment: This prints out contents of all code blocks;
        cdblsz(nocdbl); inicdb(kdbno);getcdb(kdbno);
        while (kdbno ≤ nocdbl) do
          cheokd (kdend,kdbno);
          while (¬ kdend) do
            kdtkn (kdbno,class,length,token(1:1000));
            inibuf(j);
            if (class=stnocl) then
                  begin entno ← token(1)/10;
                        fchstn (entno,istno,nduml,
                                ndum2,errflg);
                        isgn←0;
                        if (istno < 0) then isgn←1; fi
                        istno←iabs(istno);
                        sepcns(istno,ln,dig(1:20));
                        if(isgn=1) then lnl ← ln+1;
                                   else lnl ← ln;
                        fi
                        lbl ← 5-lnl;
                        if(lbl≠0) then for i=1 to lbl do
                                       entbuf(blank,
                                                 j);
                                       od
                        fi
```

```
                            if (isgn=1) then entbuf(minus,j);
                            fi
                            for i=1 to ln do entbuf(dig(i),
                                                  j); od
                            entbuf (blank,j);
                            kdtkn (kdbno,class,length,
                                   token(1:1000));
                        end
                    else for i = 1 to 6 do entbuf(blank,j);od
                fi
                klprev← 0; iflag ←'false';
                if (class=lgifcl) then iflag ←'true'; fi
                while (class≠keol) do
                   if (class=knt) then getcdb (kdbno);
                                       kdtkn(kdbno,class,
                                             length, token
                                             (1:1000));
                                       if end of line go
                                          back to the loop;
                   fi
                   if (length≠0) then for i=1 to length do
                                           entbuf(token(i),
                                                 j);
                                       od
                       else entno←token(1);
                            fetch appropriate entry and enter
                            into buffer variable name etc.;
                   fi
                   klprev← class;
                   kdtkn (kdbno,class,length,token(1:1000));
                od
                if (iflag∧klprev=rpar) then prlgif (kdbno,j,
                                                  errflg,class,
                                                  length,token
                                                  (1:1000)); fi
                enter blanks in buffer where required;
                prtbuf;
                cheokd (kdendl,kdbno);
             od
             getcdb (kdbno);
          od
end
```

```
105.          Procedure prlgif (kdbno,j,errflg,class,length,
              token (1:1000));
    begin     Comment: This routine enters target of a logical IF
                       stmt. into buffer;
              checkd (kdend,kdbno);
              if ( ¬ kdend) then error; return fi
              getcdb (kdbno);
              kdtkn(kdbno,class,length,token(1:1000));
              while (class≠keol) do
                 if (class=knt) then getcdb(kdbno);
                                     kdtkn (class,length,
                                            token(1:1000));
                                     go back to the loop beginning;
                 fi
                 if (length≠0) then for i=1 to length do
                                          entbuf (token(i),j);
                                    od
                      else entno ← token(1);
                            fetch appropriate entry and enter
                              into buffer variable name etc.;
                 fi
                 kdtkn (kdbno,class,length,token(1:1000));
              od
   end

106.          Procedure cdblsz (nocdbl);
   begin      maxflb (iflb);
              gtkdno(nocdbl,iflb);
   end

107.          Procedure prtfmt (errflg);
   begin      comment: This routine prints all format statements;
              stnosz (itblsz);
              for ii = 1 to itblsz do
                  fchstn (ii,stno,fmtflg,link,errflg);
                  if (fmtflg=1) then
                       begin sepcns (stno,ln,dig(1:20));
                             lbl ← 5-ln;
                             if (lbl≠0) then for i=1 to lbl do
                                                 entbuf(blank,j);
                                             od
                             fi
                             for i=1 to ln do
                                 entbuf (dig(i),j);
                             od
                             entbuf (blank,j);
                             for i=1 to 6 do
                                 entbuf (fmt(i),j);
                             od
                             entno ← link/10;
                             nxt ← link-entno*10;
                             if(nxt≠kmfmt) then error; return fi
                             bufcmf (entno,j);
                             enter blanks at the end of buffer;
                             prtbuf;
                       end
                  fi
    end
```

APPENDIX D

	Module Name	Procedures in Module	Caller Modules	Called Modules
1.	STRGRP	PRSGMT PRSTMT PRBLST PROCDO PRCTL PRTGT PRDECL ENTVAR ENTID PRCFUN ENTXPR GETARG ENTNMB INIDUM GETDUM	MAIN	LEXIC STACK OUTPUT COMFMT CONST DMNSON SMPVAR SUBFUN STMTNO SEGMNT FLOWBL CODEBL DECLBL
2.	LEXIC	DOLXI LEXCAL NXTCHR CHRCOD FSMTBL	STRGRP	COLLST OUTPUT RWRD
3.	COLLST	CLSTMT	LEXIC	-
4.	OUTPUT	NEWOUL NEWTKN ADDCHR DEFCLS ENDTKN ENDOUL CNSINT NEWLXI CHEOLX FCHTKN	LEXIC STRGRP	-
5.	RWRD	RESTBL STRTWD OKNXCH KODE	MAIN LEXIC	-

6. STACK	INISTK PSHSTK CHKSTK CHENDO	STRGRP	-
7. CNVTPR	PRTSEG PRTDCL PRTCD PRTFMT PRLGIF CDBLSZ BUFCMF BUFCNS BUFDMN BUFSMP BUFSFN BUFSTN SEPCNS	MAIN	OUTBUF COMFMT CONST DMNSON SMPVAR SUBFUN STMTNO SFGMNT FLOWBL CODEBL DECLBL
8. OUTBUF	INIBOF ENTBUF PRTBUF	CNVTPR	-
9. COMFMT	INICMT ENTLNE FCHLNE ENDCMT	STRGRP CNVTPR	-
10. CONST	INICNS ENTCNS FCHCNS ENDCNS	STRGRP CNVTPR	-
11. DMNSON	INIDMN ENTDMN ADJDMN FCHDMN ENDDMN SCHDMN	STRGRP CNVTPR	-
12. SMPVAR	INISMP ENTSMP FCHSMP ENDSMP	STRGRP CNVTPR	-
13. SUBFUN	INISFN ENTSFN ADJSFN FCHSFN ENDSFN SCHSFN	STRGRP CNVTPR	-

14.	STMTNO	INISTN ENTSTN ADJSTN FCHSTN ENDSTN MAXFLB STNOSZ	STRGRT CNVTPR	-
15.	SEGMNT	INISEG ADJSEG FCHSEG	STRGRT CNVTPR	-
16.	FLOWBL	INIFLB GETFLB ENTFLB NXTFLB ENDFLB CHKFLB GTKDNO	STRGRT CNVTPR	-
17.	CODEBL	INICDB GETCDB ENTCDB ENDCDB CHEOKD KDTKN	STRGRT CNVTPR	-
18.	DECLBL	INIDCL ENTDCL ENDDCL CHENDL DCLTKN	STRGRT CNVTPR	-

APPENDIX E

Procedure Name	Module in which present	Procedure Name	Module in which present
ADDCHR	OUTPUT	ENTSFN	SUBFUN
ADJDMN	DMNSON	ENTSMP	SMPVAR
ADJSEG	SEGMNT	ENTSTN	STMTNO
ADJSFN	SUBFUN	ENTVAR	STRGRP
ADJSIN	STMTNO	ENTXPR	''
BUFCMF	CNVTPR	FCHCNS	CONST
BUFCNS	''	FCHDMN	DMNSON
BUFDMN	''	FCHLNE	COMFMT
BUFSFN	''	FCHSEG	SEGMNT
BUFSMP	''	FCHSFN	SUBFUN
BUFSTN	''	FCHSMP	SMPVAR
CDBLSZ	''	FCHSTN	STMTNO
CHENDL	DECLBL	FCHTKN	OUTPUT
CHENDO	STACK	FSMTBL	LEXIC
CHEOKD	CODEBL	GETARG	STRGRP
CHEOLX	OUTPUT	GETCDB	CODEBL
CHKFLB	FLOWBL	GETDUM	STRGRP
CHKSTK	STACK	GETFLB	FLOWBL
CHRCOD	LEXIC	GTKDNO	''
CLSTMT	COLLST	INIBUF	OUTBUF
CNSINT	OUTPUT	INICDB	CODEBL
DCLTKN	DECLBL	INICMT	COMFMT
DEFCLS	OUTPUT	INICNS	CONST
DOLX1	LEXIC	INIDCL	DECLBL
ENDCDB	CODEBL	INIDMN	DMNSON
ENDCMT	COMFMT	INIDUM	STRGRP
ENDCNS	CONST	INIFLB	FLOWBL
ENDDCL	DECLBL	INISEG	SEGMNT
ENDDMN	DMNSON	INISFN	SUBFUN
ENDFLB	FLOWBL	INISMP	SMPVAR
ENDOUL	OUTPUT	INISTK	STACK
ENDSFN	SUBFUN	INISTN	STMTNO
ENDSMP	SMPVAR	KDTKN	CODEBL
ENDSTN	STMTNO	KODE	RWRD
ENDTKN	OUTPUT	LEXCAL	LEXIC
ENTBUF	OUTBUF	MAXFLB	STMTNO
ENTCDB	CODEBL	NEWLXI	OUTPUT
ENTCNS	CONST	NEWOUL	''
ENTDCL	DECLBL	NEWTKN	'
ENTDMN	DMNSON	NXTCHR	LEXIC
ENTFLB	FLOWBL	NXTFLB	FLOWBL
ENTID	STRGRP	OKNXCH	RWRD
ENTLNE	CMF[illegible]	PRBLST	STRGRP
ENTNMB	STRGRP	PRCFUN	''

PRCTL	STRGRP
PRDECL	"
PRLGIF	CNVTPR
PROCDO	STRGRP
PRSGMT	"
PRSTMT	"
PRTBUF	OUTBUF
PRTCD	CNVTPR
PRTDCL	"
PRTFMT	"
PRTGT	STRGRP
PRTSEG	CNVTPR
PSHSTK	STACK
RESTBL	RWRD
SCHDMN	DMNSON
SCHSFN	SUBFUN
SETCNS	CNVTPR
STNOSZ	STMTNO
STRTWD	RWRD

```
$IBLDR SUBR2    05/12/77                                                        SUBR0001
$IBLDR SUBR3    05/12/77                                                        SUBR0001
$IBLDR SUBR5    05/12/77                                                        SUBR0001
$IBLDR SUBR6    05/12/77                                                        SUBR0001
$IBLDR SUBR7    05/12/77                                                        SUBR0001
$IBLDR SUBR9    05/12/77                                                        SUBR0001
$IBLDR SUBR10   05/12/77                                                        SUBR0001
$IBLDR SUBR12   05/12/77                                                        SUBR0001
$IBLDR SUBR14   05/12/77                                                        SUBR0001
$IBLDR SUBR15   05/12/77                                                        SUBR0001
$IBLDR SUBR17   05/12/77                                                        SUBR0001
$IBLDR SBCD     05/13/77                                                        SBDC0001
$IBLDR SUB5     05/12/77                                                        SUB50001
$IBLDR SUB9     05/16/77                                                        SUB90001
$IBLDR SUB10    05/16/77                                                        SUB10001
$IBLDR SUB11    05/16/77                                                        SUB10001
$IBLDR SUB15    05/16/77                                                        SUB10001
$IBLDR SUB17    05/16/77                                                        SUB10001
$IBLDR MINU     05/12/77                                                        MINU0001
$IBLDR JYOTI    05/12/77                                                        JYCT0001
$IBLDR MEENA    05/12/77                                                        MEEN0001
$IBLDR PRIYA    05/12/77                                                        PRIY0001
$ENTRY          MAIN
       CSG029                                           IBLDR -- JOB    000000
```

A SAMPLE OUTPUT

```
                        M E M O R Y   M A P

SYSTEM, INCLUDING IOCS                                  00000 THRU  12251
FILE BLOCK ORIGIN                                       12260
     NUMBER OF FILES -       2
     1.   S.FBIN                       12260
     2.   S.FBOU                       12303
OBJECT PROGRAM                                          12326 THRU  76525
     1.   DECK 'MAIN  '                12326
     2.   DECK 'JACK  '                12561
     3.   DECK 'SUB14 '   *            20647
     4.   DECK 'SUB1  '   *            21140
     5.   DECK 'SUB24 '   *            23631
     6.   DECK 'SB20  '                26007
     7.   DECK 'SUR   '                26344
     8.   DECK 'SUB12 '                27403
     9.   DECK 'SUB18 '   *            30770
    10.   DECK 'SUB8  '                31230
```

```
11.  DECK 'SUB3  '  *        31457
12.  DECK 'SUB6  '  *        34115
13.  DECK 'SUBR13'           35421
14.  DECK 'SUB2  '           37335
15.  DECK 'SUBR11'  *        40633
16.  DECK 'SUB4  '  *        41051
17.  DECK 'SB3   '           42252
18.  DECK 'SUB13 '           42571
19.  DECK 'SUB7  '           43066
20.  DECK 'PRNT  '  *        43326
21.  DECK 'SUBR20'  *        54107
22.  DECK 'SUBR16'  *        54144
23.  DECK 'SUBR19'  *        54274
24.  DECK 'SB17  '  *        54325
25.  DECK 'SB19  '  *        54553
26.  DECK 'SUB19 '  *        56040
27.  DECK 'SUB20 '  *        56062
28.  DECK 'SUB22 '  *        56262
29.  DECK 'SUB23 '  *        56313
30.  DECK 'SUB26 '  *        56335
31.  DECK 'SUBR21'  *        56366
32.  DECK 'SUBR22'  *        57012
33.  DECK 'SUBR23'  *        57035
34.  DECK 'SUBR24'  *        57110
35.  DECK 'SUBR25'  *        57157
36.  DECK 'SUMA  '  *        57266
37.  DECK 'SUJA  '  *        57336
38.  DECK 'VANI  '  *        57411
39.  DECK 'MONA  '  *        57434
40.  DECK 'JILL  '  *        57464
41.  DECK 'SUB16 '  *        57523
     CSG029                                   IBLDR -- JOB   000000

42.  DECK 'SB1   '  *        57607
43.  DECK 'SB2   '  *        65164
44.  DECK 'SB4   '  *        65244
45.  DECK 'SB5   '  *        65503
46.  DECK 'SB6   '  *        65525
47.  DECK 'SB7   '  *        65562
48.  DECK 'SB8   '  *        65614
49.  DECK 'SB9   '  *        65643
50.  DECK 'SB10  '  *        65673
51.  DECK 'SB11  '  *        65721
52.  DECK 'SB12  '  *        66060
53.  DECK 'SB13  '  *        66102
54.  DECK 'SB14  '  *        66136
55.  DECK 'SB15  '  *        66223
56.  DECK 'SB16  '  *        66473
57.  DECK 'SB18  '  *        66646
58.  DECK 'SUBR1 '  *        66743
59.  DECK 'SUBR2 '  *        66765
60.  DECK 'SUBR3 '  *        67205
61.  DECK 'SUBR5 '  *        67307
62.  DECK 'SUBR6 '  *        67340
63.  DECK 'SUBR7 '  *        67362
64.  DECK 'SUBR9 '  *        67603
65.  DECK 'SUBR10'  *        67634
66.  DECK 'SUBR12'  *        67656
67.  DECK 'SUBR14'  *        70145
68.  DECK 'SUBR15'  *        70176
69.  DECK 'SUBR17'  *        70220
70.  DECK 'SBDD  '           70301
71.  DECK 'SUB5  '           71214
72.  DECK 'SUB9  '           71702
73.  DECK 'SUB10 '           71725
74.  DECK 'SUB11 '           71753
75.  DECK 'SUB15 '  *        72215
76.  DECK 'SUB17 '  *        72236
77.  DECK 'MINU  '  *        72266
78.  DECK 'JYOTI '  *        72310
79.  DECK 'MEENA '  *        72445
```

```
 80.  DECK 'PRIYA '  *          72576
 81.  SUBR 'INSYFB'             72730
 82.  SUBR 'OUSYFB'             72767
 83.  SUBR 'POSTX '             73020
 84.  SUBR 'CNSTNT'  *          73331
 85.  SUBR 'FPR   '             73341
 86.  SUBR 'FRC   '             73342
 87.  SUBR 'IOS   '             73343
 88.  SUBR 'RWC   '             73622
 89.  SUBR 'ACV   '             74776
 90.  SUBR 'HCV   '             75070
 91.  SUBR 'ICV   '             75173
 92.  SUBR 'XCV   '             75213
 93.  SUBR 'INTJ  '             75231
 94.  SUBR 'FPT   '             75545
 95.  SUBR 'XEM   '  *          76161
    CSG029                                        IBLDR -- JOB   000000

  (* - INSERTIONS OR DELETIONS MADE IN THIS DECK)

INPUT - OUTPUT BUFFERS                              76637 THRU  77776

UNUSED CORE                                         76526 THRU  76631

                   *** OBJECT PROGRAM IS BEING ENTERED INTO STORAGE AT  11 HRS.
```

```
      REAL I,J,K
C     TO FIND THE BIGGEST OF THREE NOS.
      READ 1,I,J,K
    1 FORMAT(3F8.5)
      BIG=I
      IF(J.GT.I) GO TO 10
      IF(K.GT.I) GO TO 5
       PRINT 2,I
      STOP
    5 PRINT 2,K
      STOP
   10 IF(K.GT.J) GO TO 5
      PRINT 2,J
      STOP
    2 FORMAT(1H0,5X,F15.8)
      END
```

SEGMENT HEADER

COJ$

COMMENT AND FORMAT TABLE

13	72	C TO FIND THE BIGGEST OF THREE NOS.
-40	60	(3F8.5)
-40	60	(1H0,5X,F15.8)

CONSTANT TABLE

1

2

2

2

DIMENSION TABLE

SIMPLE VARIABLE TABLE

I	1	1
J	1	1
K	1	1
BIG	0	0

SUBROUTINE/FUNCTION TABLE

STMT. NO. TABLE

		-9999	0
1	1	21	
-9999	0	2	
10	0	7	
-9999	0	3	
-9999	0	4	
5	0	6	
-9999	0	5	
-9999	0	8	
-9999	0	9	
2	1	31	

FLOW BLOCK NO. 1
****

8

0

-9999

1

36

56

****

FLOW BLOCK NO. 2
****

7

0

-9999

2

46

****

FLOW BLOCK NO. 3
****

8

0

-9999

3

66

86

****

FLOW BLOCK NO. 4
****

7

0

-9999

4

76

****

FLCW BLOCK NO. 5
****

7

0

‾9999

5

0

****

FLCW BLOCK NO. 6
****

7

0

5

6

0

****

FLCW BLOCK NC. 7
****

8

0

10

7

96

106

****

FLCW BLOCK NO. 8
****

7

0

-9999

8

76

```
********************


FLOW BLOCK NO.  9
********************

          7

          0

      -9999


          9

          0

********************


CODE BLOCK NO.   1
*******************************************

  -60          8     CONTINUE

   10          1     $

   13          0             11

   10          1     $

  -60          4     READ

    1          0             12

    9          1     ,

    0          0             14

    9          1     ,

    0          0             24

    9          1     ,

    0          0             34

   10          1     $

    0          0             44

    6          1     =

    0          0             14

   10          1     $

  -70          2     IF

   -7          1     (

    0          0             24

   -1          4     .GT.

    0          0             14

    8          1     )
```

```
  10          1      $

********************************************

CODE BLOCK NO.   2
********************************************

 -81          4      GOTO

   1          0              46

  10          1      $

********************************************

CODE BLOCK NO.   3
********************************************

 -60          8      CONTINUE

  10          1      $

 -70          2      IF

   7          1      (

   0          0              34

  -1          4      .GT.

   0          0              14

   8          1      )

  10          1      $

********************************************

CODE BLOCK NO.   4
********************************************

 -81          4      GOTO

   1          0              76

  10          1      $

********************************************

CODE BLOCK NO.   5
********************************************

  60          8      CONTINUE

  10          1      $

 -60          5      PRINT

   1          0              22
```

```
     0          0               14

    1C          1     $

   -82          4     STCP

    1C          1     $

 ****************************************

 CODE BLOCK NO.    6
****************************************

    14          0               76

   -6C          5     PRINT

     1          0               32

     9          1     ,

     C          0               34

    1C          1     $

   -82          4     STOP

    1C          1     $

 ****************************************

 CODE BLOCK NC.    7
****************************************

    14          0               46

   -7C          2     IF

     7          1     (

     C          0               34

    -1          4     .GT.

     C          0               24

     8          1     )

    1C          1     $

 ****************************************

 CODE BLOCK NO.    8
****************************************

   -81          4     GOTO

     1          0               76

    1C          1     $

 ****************************************
```

CODE BLOCK NC. 9
****

-6C	8	CCNTINUE	
1C	1	$	
-6C	5	PRINT	
1	0		42
9	1	,	
C	0		24
1C	1	$	
-82	4	STOP	
1C	1	$	

****

DECLARATIVE BLOCK

****

-21	4	REAL	
0	0		14
9	1	,	
0	0		24
9	1	,	
0	0		34
10	1	$	

****

JOB STATISTICS -

CVERFLOWS	COCCOO
UNDERFLOWS	COCCOO
CARDS READ	COCC53
LINES PRINTED	COC285
CARDS PUNCHED	COCCOO
FORTRAN SUB. ERRORS	COCCOO
